SigMath User Guide

An Introduction

by
Michael Adam, Georg Berschneider, Daniel Karrasch,
Zoltán Sasvári, Szilvia Vock

v1.0
November 2, 2011

Contents

List of Figures **7**

List of Tables **9**

1 Introduction **11**
1.1 What is SigMath? . . . 11
1.2 Basic principles . . . 12
1.3 Installation . . . 13
1.4 Getting started . . . 13
1.5 The editor . . . 14
1.6 SigMath documents . . . 15
1.6.1 Basic structure . . . 15
1.6.2 Text, commands, and comments . . . 17
1.6.3 Inserting documents . . . 18

2 Word processing **21**
2.1 Environments . . . 21
2.2 Page layout . . . 21
2.3 Fonts . . . 23
2.4 Formatting . . . 25
2.5 Positioning of text . . . 27
2.6 Enumerations . . . 29
2.7 Inserting figures . . . 30
2.8 Colors and background . . . 32
2.9 Tables . . . 33
2.10 Symbols . . . 37

3 User interaction **39**
3.1 Hyperlinks . . . 39
3.2 Input fields . . . 42
3.3 Input vectors and input matrices . . . 43
3.4 Event handling . . . 45
3.5 Draw and move selected objects . . . 48

4 Plots in SigMath **51**
4.1 Two-dimensional plots . . . 51
4.1.1 Generating plots . . . 51

4.1.2 Color, style, and background . . . 55
4.1.3 Redrawing plots . . . 57
4.1.4 Axes . . . 60
4.1.5 Interactive plots . . . 62
4.1.6 Text . . . 63
4.1.7 Summary . . . 63
4.2 Three-dimensional plots . . . 63

5 Exporting graphics 65

6 Programming 71
6.1 Classes . . . 71
6.2 Data types . . . 73
6.2.1 Constants . . . 74
6.2.2 Variables . . . 74
6.2.3 Elementary data types . . . 78
6.2.4 Operators and expressions . . . 80
6.3 Complex data types . . . 84
6.3.1 Strings . . . 84
6.3.2 Arrays . . . 92
6.3.3 GLPoint . . . 99
6.3.4 Container . . . 102
6.3.5 Bitmaps . . . 105
6.3.6 Text editors . . . 105
6.4 Control structures . . . 105
6.4.1 Branches . . . 106
6.4.2 Loops . . . 109
6.5 Functions . . . 112
6.6 Multithreading . . . 114
6.7 Debugging . . . 117

7 Computer graphics 121
7.1 What is OpenGL? . . . 121
7.2 Primitive . . . 121
7.2.1 Vertices . . . 122
7.2.2 OpenGL primitives . . . 122
7.2.3 Implementation of primitives . . . 123
7.2.4 Points and lines . . . 124
7.2.5 Surfaces . . . 125
7.2.6 Additional functions . . . 129
7.2.7 Line properties . . . 129
7.2.8 Quadrics . . . 132
7.2.9 An extended example . . . 134

7.3 Working with lists . . . 134
7.3.1 Generating lists . . . 134
7.3.2 Calling lists . . . 134
7.4 Transformations . . . 135
7.4.1 Scaling . . . 135
7.4.2 Translation . . . 136
7.4.3 Rotation . . . 136
7.5 OpenGL matrices . . . 136
7.5.1 Functionality of OpenGL matrices . . . 137
7.5.2 Matrix stack . . . 137
7.6 Scenery . . . 138
7.6.1 Positioning of the camera . . . 139
7.6.2 Projection . . . 139
7.6.3 Viewport . . . 141
7.7 Textures . . . 141
7.7.1 Basic principles . . . 141
7.7.2 Types of textures . . . 141
7.7.3 Loading textures . . . 142
7.7.4 Mipmapping . . . 147
7.7.5 Texture mapping . . . 147

8 Using and Writing DLLs **151**
8.1 Loading DLLs . . . 151
8.2 Writing DLLs . . . 153

9 Rendering **159**

Index **161**

List of Figures

1.1 Starting up SigMath 14
1.2 Editor window 15

2.1 The page layout template "Standard (with date)/multipage" 22
2.2 Effect of the page format parameters 24
2.3 Positioning of text using `TextPos` 29
2.4 Example for a color gradient 34

3.1 Four squares and mouse motion 50

4.1 The sine function on the interval $[0, 2\pi]$ 53
4.2 Plotting a circle in two different ways. 54
4.3 `PlotStyle` options 56
4.4 Redrawing plots depending on user interaction 59
4.5 Placing tick marks manually 62

5.1 The graphics export GUI 66

7.1 Lines with different line patterns. 131

List of Tables

1.1 Menu items in the editor . . . 16
1.2 Shortcuts in the editor . . . 17
1.3 Commands, that return file locations . . . 19

2.1 Environments in SigMath . . . 22
2.2 Font scaling commands . . . 25
2.3 Shorthands for horizontal spacing . . . 27
2.4 Commands for colors . . . 33
2.5 Symbols in SigMath . . . 38

3.1 Messages for input arrays . . . 44
3.2 WINDOWS system messages . . . 46
3.3 Functions returning ASCII code of special keys . . . 46
3.4 Mouse cursor . . . 48

4.1 Plot properties . . . 64

5.1 Menu items of the graphics export dialog window . . . 66
5.2 Supported file formats in the graphics export . . . 67
5.3 Font families for the graphic export . . . 68
5.4 Advanced options in graphics export . . . 69

6.1 Classes in SigMath . . . 72
6.2 Constants . . . 74
6.3 Data types in SigMath . . . 75
6.4 Reserved words in SigMath . . . 76
6.5 Ranges for variables of type `Integer`, `Real`, and `Complex` . . . 78
6.6 Operators in SigMath . . . 81
6.7 Automatic type conversion for arithmetic operators . . . 81
6.8 Truth tables . . . 83
6.9 Determination of the base for `Integer` conversion . . . 92
6.10 Available array types in SigMath . . . 93
6.11 Container types in SigMath . . . 103

7.1 Types of primitives . . . 123
7.2 provided functions . . . 150

8.1 Possible Suffices for DLL functions and properties . . . 153

1 Introduction

In this section we want to introduce the basic functionalities of SigMath and show the first steps to create a SigMath document. In later sections we give more details on the different functions of SigMath and provide examples to nearly all features. Most of the examples are also available as files than can be loaded and tested in SigMath. The filenames are indicated in brackets.

1.1 What is SigMath?

SigMath is an application which allows to visualize interactive content in an easy way. SigMath works as interpreter of its own programming language and combines the following functionality:

- graphics and animation
- text layout on the screen
- user interaction
- programming techniques
  (from basic commands to complex data types and threading)

Interactive functions allow user interaction and navigation in SigMath documents as known from web pages. SigMath's graphical engine is based on OpenGL. Each document interpreted by SigMath as an animation, i.e., the content of the document is permanently updated and redrawn. This allows to perform user-driven mathematical computations and visualize its results in "real time". The text layout syntax is similar to LaTeX syntax. Whereas LaTeX aims at producing printed, and therefore static, documents, SigMath's layout capabilities are designed for dynamic text display on the screen. Additionally, graphic export is available based on the GL2PS–library by C. Geuziane.

For developers SigMath provides a highly-developed programming language which not only contains the usual control structures and complex data types but also allows the usage of OpenGL commands, Windows system routines and multithreading. SigMath's functionality is easily extendable by importing functions and methods written in C/C++.

Summing up, SigMath allows to generate animations, performs the necessary mathematical computations, displays text and enables the user to interact and navigate through the animation. Although there might be applications designed for every single task, we think that the mixture provided by SigMath makes it a powerful tool.

1.2 Basic principles

All applications, projects or presentations in SigMath consist of *documents*, where a document is a plain text file (in ASCII format). These files contain *source code*, which tells SigMath what to show, and how it should be presented. If we open a SigMath document in SigMath the source code is automatically "interpreted" and the result is shown on the screen. If the code only contains text, it is displayed on the screen as it is. In order to present formatted text, additional commands have to be included in the source code. On the difference of text and commands see Section 1.6.2.

Every document in SigMath is displayed as an animation. That means, a lot of images – every one static – is displayed in high speed and gives the illusion of movement. This is accomplished by parsing the current page in an (invisible) infinite loop and displaying it after every run. The refreshment rate of the presentation is given in fps (frames per second). Its actual value depends on several factors:

- how fast are central processing unit and graphics card;
- what computations are contained in the current page;
- the frame rate of the monitor;
- what other programs are running in the background.

If the refreshment rate is too low, we detect jumps in the presentation. For the human eye, a rate of 30 fps gives the impression of a rather smooth visualization. Between the frames, user interaction is processed. Thus, at a frame rate of 0.1 fps, a user has to wait 10 seconds on processing a mouse click.

This way of interpreting the source code might entail unforeseen effects for programmers not used to animation. We illustrate these effects by a small pseudo-example. The task is to add 5 to a user-specified number and show the result on the screen. A first attempt would look the following:

Code: (Pseudo program)

Request a user-specified number x	`\Input("x");`
$x = x + 5$	`\x += 5;`
Display x	`\ValueOf("x");`

Only a small portion of programmers would define a new variable in the second line, e.g., by `y = x + 5`, since `x` is not used later on and the current value need not be stored. The animation-oriented interpretation in SigMath leads to the following: in every frame 5 is added to `x`, since almost all source code is processed in an infinite loop. The displayed value on the screen increased more and more without any user interaction. That is not, what the program was intended to do!

This little example should give you an idea that not all familiar programming concepts lead to the desired results. This includes self-referencing commands as in the example that should – at least – be handled with care. The advantages are clear: the programmer does not have to deal

with repetitions of his source code and can even make use the effect shown in the pseudo-example.

1.3 Installation

SigMath comes as a zip-archive. In order to install a new version of SigMath remove an older existing version by deleting the SigMath program folder before unzipping the new version. The following instructions are valid for the operating systems WINDOWS 98/2000/NT/XP and WINDOWS VISTA:

1. Download the file `SigMathZIP.exe` from the SigMath-web page

   http://www.math.tu-dresden.de/~sasvari/sigmath/

2. Start the executable file by double clicking it. You are asked for the target directory.

   Choose a path and click on "Extract"

3. Open Explorer, choose the SigMath-directory in the path and right-click the file

   `SigMath.exe` and drag it to the desktop. After releasing choose "Create Link".

   Double clicking the icon on the desktop will start SigMath.

Small SigMath examples that are used in the following sections can be found in the folder `SigMath-Guide`, which can be downloaded from the above mentioned web page.

1.4 Getting started

Before we show how to layout your own SigMath documents we want to make you familiar with the SigMath development tools. We show how to navigate in SigMath and to use already built up sample documents. After starting SigMath the user interface as depicted in Figure 1.1 is shown.

The top line shows the version number of the running SigMath application on the right and on the elements "Deutsch" and "English" which switch the system to German or English. The bottom line contains the current date and time. In the middle of the page several menu elements are listed. Selecting a menu item by left-clicking executes connected commands. One can open a SigMath document, start the editor to write and compile a new document, or change general options. Clicking on "Close" or hitting the *ESC* key quits the program.

The green words on the SigMath start page are *interactive elements* (see Chapter 3) and are links to new pages (see Section 3.1). These elements allow the user/viewer to interact with SigMath. They can be used to:

1. direct to a new page (by left-clicking them);
2. execute one (or several) commands (by left-clicking them);
3. change the value of a variable: hold the right mouse button and move up or down

   OR left-click the element; then enter a new value in the appearing input frame

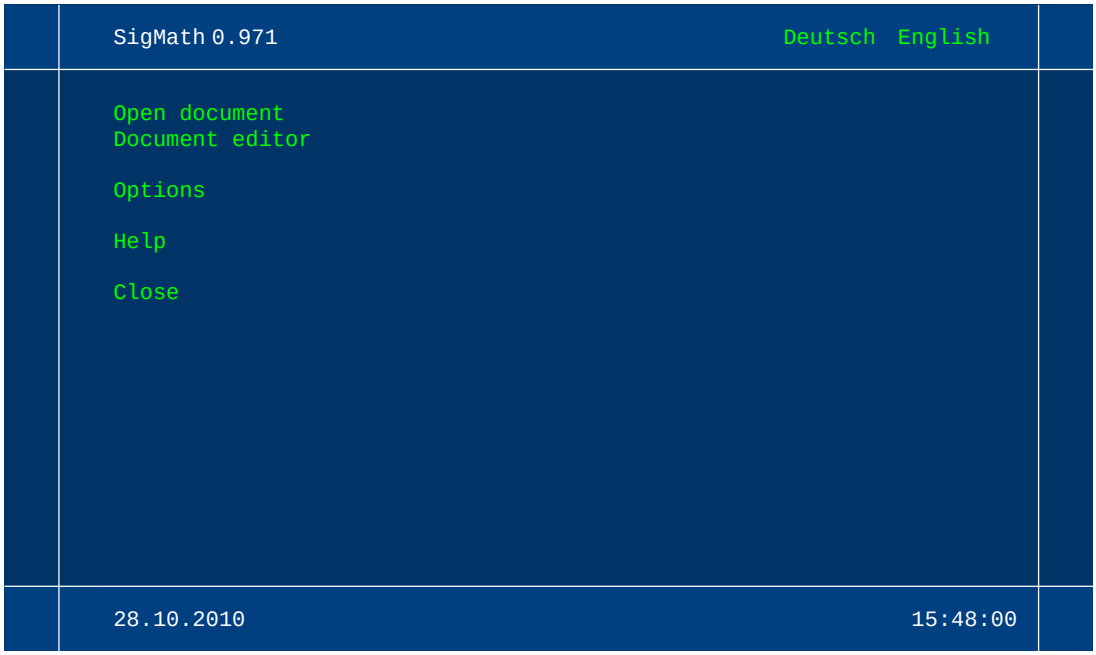


Fig. 1.1: Starting up SigMath

(remove previous value by using the *DEL* key);

4. move objects (press left mouse button and move the mouse).

Navigation within documents is similar to navigation on web pages. By clicking input fields one can change pages. Navigation via keyboard involves the following keys:

- *ESC* closes the current page if the documents allows it.
- Using the cursor-keys or mouse wheel one can move text that is not fixed to a certain position.
- If the page contains hidden elements (e.g., for a presentation), they can be shown by *PageUp* and *PageDown*.

If an animation is running on the screen, it can be stopped by pressing *CTRL + Space* simultaneously, if the documents allows it. For example, the time on the start page, which is an animation, can be stopped in this way. Hitting *CTRL + Space* again or left-clicking in the window the animation is resumed.

1.5 The editor

Since SigMath documents are platform-independent text files, basically every text editor can be used to write SigMath source code. Nevertheless it might be worth considering to use the built-in text editor. By a simple mouse click a SigMath document can be compiled and run, directly. On the SigMath start page (Figure 1.1) the editor opens up by left-clicking ”Editor”. Then the user interface depicted in Figure 1.2 is displayed.

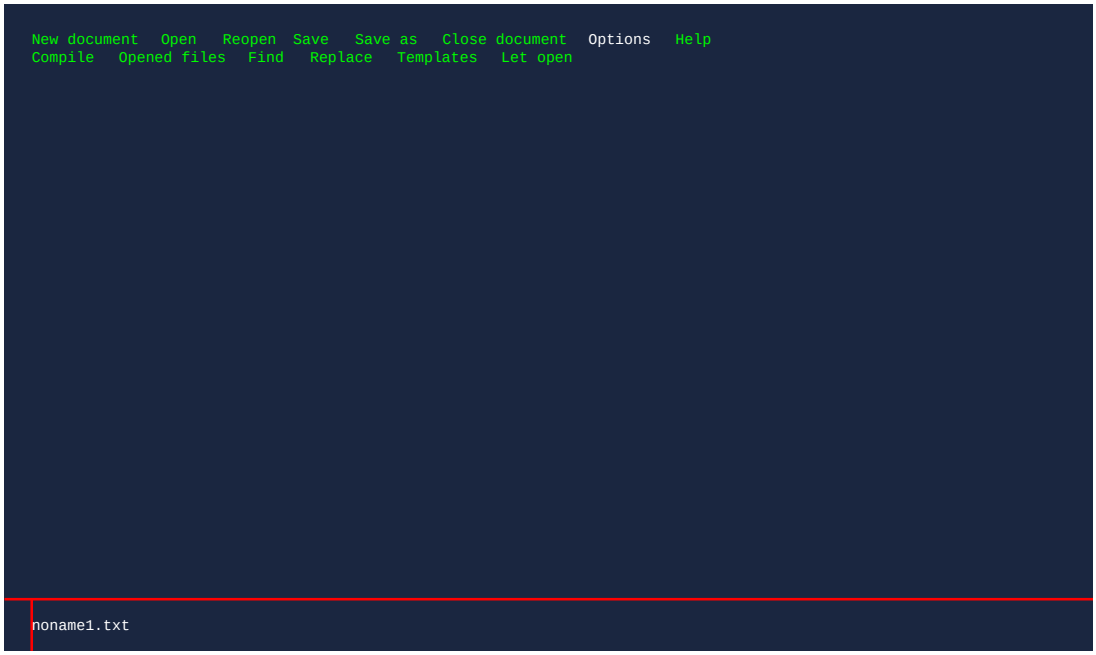

New document Open Reopen Save Save as Close document Options Help
Compile Opened files Find Replace Templates Let open
noname1.txt

Fig. 1.2: Editor window

In this editor SigMath documents can be created and compiled. Table 1.1 lists the menu points and their functions. The editor provides keyboard shortcuts for improving work flow. Shortcuts are key combinations connected to frequently used functions. They are listed in Table 1.2.

Another important feature of the editor is that it allows to place anchors or marks in the text. Anchors are labels on lines which can be referenced to. If one is working on a document containing a lot of source code and one has to modify a certain text passage repeatedly, one can use an anchor to ”jump” to this text passage. Anchors are placed by using the key combination *CTRL + Shift + Number*. Any number between 0 and 9 is possible. The number appears then on the left margin of the line. To jump to this anchor one uses *CTRL + Number*. To remove the anchor move the cursor to the corresponding line (e.g., by clicking on the line) and hit *CTRL + Shift + Number*.

The editor quits by hitting *ESC* and one returns to SigMath’s start page.

1.6 SigMath documents

1.6.1 Basic structure

We illustrate the basic structure of SigMath documents by a simple example. Let us open the text editor in SigMath’s user interface.

Every document consists of a preamble and a main part. The preamble may contain the definition of globally accessible variables or constants. All commands in the preamble are only executed once, i.e., they are excluded from the previously mentioned infinite loop. The main part contains those informations, text and commands that are executed and displayed repeat-

Menu point	Functionality
New document	Opens a new (empty) document.
Open	Opens a selected file.
Reopen	Shows a list of recent files. Clicking on a filename reopens the file in the editor.
Save	Saves the document using the last specified path and filename.
Save As	Saves the document to the specified path using a (maybe different) filename.
Close Document	Closes the current document.
Help	Displays a description of the editor. Topics can be selected in the menu. Select a menu item by left-clicking.
Compile	Saves and compiles the current document.
Opened Files	Shows a list of currently opened documents in the editor. By left-clicking the filename the content of the file is shown.
Find	Specify the text to find by clicking "Text". Searches the document starting at the current position of the cursor or starting from the beginning of the file (Select by clicking)
Templates	Inserts a selected template after the current cursor position. The template can be selected by left-clicking.
Let Open	Opens the selected document in the default application specified by the operating system

Tbl. 1.1: Menu items in the editor

edly. It starts, known to LaTeX users, with the command `\begin{document}` and ends with the command `\end{document}`. Thus, the simplest SigMath documents consists of the following two lines:

```
\begin{document}

\end{document}
```

If we compile this document, SigMath will show a black screen. To add more content to this document, we can place arbitrary text in between the two command lines. Since we have not treated commands yet, this text may consist of ordinary characters like literals, numbers and punctuation marks. Just take the classic: Hello World! Since text is not bound to any position in SigMath by default, it can be moved using the arrow keys upwards/downwards, to the left and to the right. But, not only the text is moved but actually the whole page moves accordingly. With the following lines of code the text line "Hello World!" is initially shown in white letters on the top left corner of a black screen.

Shortcut	Action
CTRL + A	Change view between current and last viewed document
CTRL + X	Cut text
CTRL + C	Copy text
CTRL + V	Paste text
CTRL + F	Save and compile the current document
CTRL + P	Declare current document as master document of a larger project
CTRL + S	Save the document
CTRL + Z	Undo the last edit
DEL	Delete the symbol after the cursor
End	Jump to the end of the current line
CTRL + End	Jump to the last line of the document
Home	Jump to beginning of current line
CTRL + Home	Jump to the first line

Tbl. 1.2: Shortcuts in the editor

Code: (HelloWorld.txt)
```
\begin{document}
 Hello World!
\end{document}
```

1.6.2 Text, commands, and comments

If we only insert plain *text* in the document it will be displayed in white letters using a default font (RM) and font size (small) on a black screen. The actual size depends on the size of the monitor and the specified graphics resolution. *Commands* can be used at any point in code. Commands are denoted by \ and thereby detected by the interpreter (the "compiler" of the source code in a SigMath document). Several commands can be grouped using the `\code` command:

```
\code{ here arbitrary many commands without using "\" }
```

Everything contained within the curly brackets is regarded as a command, so \ has to be omitted.

To improve readability of code one should include comments to the source code. It can be used to explain the meaning of certain variables, commands or even of a part of code. This helps other programmers to understand what you want to do or even helps yourself if you want to understand what you intended to do when writing the code.

In SigMath we have to distinguish comments within the text regions and the command regions of the document. Command regions are, e.g., code blocks or functions. Comments in the text

may be embedded as in LaTeX by using `%`. This character tells SigMaths' interpreter to ignore the rest of the line. For commands in the command regions, we use the C/C++ comment symbol:

- everything written in between `/*` and `*/` is ignored,
- and `//` turns the rest of the line into a comment.

This distinction can be used to "hide" a lot of consecutive lines in the text region from the interpreter by commenting them. We add a code section and put `/*` at the beginning and `*/` at the end of the section.

```
\code{
/*
⋮
text, that should be hidden as a comment
⋮
*/
}
```

1.6.3 Inserting documents

If documents contain a lot of source code, the contents tends to become confusing. Thus, one should split the document into several smaller portions and connect them using the `Include` command. Thus way also allows to create and save templates for certain tasks that can be used over and over again. The text file is included at the position of the `Include` command in the document and only executed on compilation.

Syntax: `\Include(filename);`

The `Include` command requires the name of a text file `filename` as its argument. In order to make this work, one has to specify the correct path. Table 1.3 contains a number of commands that make the task easier. If the file to be included is contained in the same directory as the current document, we write

```
\Include(DocDir + "filename.txt");
```

If we want to include the file `Include.txt` which is located in a folder `IncludeFiles` in the directory `Examples` we have to specify the path in the following way

```
\Include(DocDir + "\Examples\IncludeFiles\Include.txt");
```

The commands `DocDir`, `DocName`, `ProgDir`, and `HelpDir` are used as in the examples. For navigation in upper level directories we use the command `syst.ParentDir`.

Syntax: `\syst.ParentDir(level,dirName);`

`ParentDir` requires two arguments: the directory `dirName`, we start with and the number of levels `level`, we want to move up in the directory tree. The argument `level` has to be a non-negative integer. The level 0 returns `dirName` itself. We will illustrate the usage of `Include` commands when we learn more on page layout templates in Section 2.2.

Command	**Directory**
`DocDir`	Directory that contains the current document
`DocName`	full qualified path name (directory and filename) of the current document
`ProgDir`	Directory that contains the executable `SigMath.exe`
`HelpDir`	the help-directory (depends on the chosen language version)
`ParentDir`	parent directories

Tbl. 1.3: Commands, that return file locations

A second variant of the `Include` command accepts as an argument the name of a text block. To do this, one has to define a text block and "fill" it:

Syntax:
```
\begin{#textBlockName}
 Text and commands
\end#
```

Later the content of this text block can be inserted using its key `#textBlockName`

```
\Include(#textBlockName);
```

at this point.

Another, rather unusual way of including code is given by the `IncludeString` command.

Syntax: `\IncludeString(stringVariable);`

The argument is a text variable, that contains the source code. This allows to generate code fragments on run time which may depend on user interaction or the values of other variables and to include the code at this position.

Example 1.1 Let us write a program that performs the rather dull job of displaying all numbers from 1 to 1000 below each other. In a rather naive attempt one would actually write the 1000 lines of code. Typically, in programming languages one would use a loop structure (more on loops in Section 6.4.2) to perform this job. In this introduction we stick to pseudo-code and

provide SigMath code on the right hand side.

Code:

```
S = ""                                                    S = "";
From i=1 to 1000 repeat                                 for(i=0,i<=1000,i++){
 append to S the following number                         S += i + string.n};
 and the command for a newline
```

As a last remark, the task in this little example also can be solved using the `array` environment (see Section 2.9).

In a similar manner as `Include` and `IncludeString` the commands `Execute` and `ExecuteString` behave.

Syntax:

```
\Execute(filename);
\ExecuteString(stringVariable);
```

Here the contents of the file `filename` or the variable `stringVariable` is assumed to be source code. Its content is treated as we would have written it within a `code` environment (see Section 1.6.2).

2 Word processing

This chapter is devoted to questions of word processing, i.e., layouting and positioning of text. LaTeX users will recognize a lot of the features in SigMath. In fact, they were in the focus of the developers of SigMath. Nevertheless, the experienced LaTeX user should be aware of some subtilities that arise from the way SigMath presents text.

If you want to write several documents we recommend to generate one file only containing the layout specifications and include this file to the preamble of all text documents using the `Include` command (see section 1.6.3). Furthermore, layout specifications only apply to the current , i.e. the environment they are contained in (see section 2.1). within.

2.1 Environments

Before we treat more concise details of the word processing features of SigMath let us introduce the abstract notion of an enviroment. Environments are used to group several commands or text and to separate it from the rest of the document. As known from, e.g., LaTeX, enviroments are given in the following form:

Syntax:

```
\begin{NameOfEnvironment}
 some text or commands
\end{NameOfEnvironment}
```

Everything within such an environment is processed according to certain specified parameters. Thuis, text within the enviroments may appear in different color, font size, or font family than other text in the document. Very simple enviroments are given in the following form:

text outside { \newTextColor *text inside the environment* } *text outside*

In this example, the *text outside* appears in another text color as the *text inside*.

We list all currently implemented environments in SigMath in Table 2.1.

2.2 Page layout

In this section we introduce how to layout pages in SigMath at first without dealing with content that appears on the page later on.

Name of the environment	Description	Reference
`array`	Tables and matrices	Section 2.9
`centerline`	Centering text	Section 2.4
`document`	document environment	Section 1.6.1
`enumerate`	Listings	Section 2.6
`env`	environments for internal use	Page 68
`inputlabel`	formatted text in links	Page 40
`plot2d`	2D-graphics	Section 4.1
`plot3d`	3D-graphics	Section 4.2
`postext`	Positioning of text	Section 2.5

Tbl. 2.1: Environments in SigMath

The simplest page layout we encoutered was a blank page with white text on black background (see Section 1.6.1). Some predefined (simple) page layouts can easily be used by inserting the corresponding layout template which can be found in the menu item ”Layout” of the ”Template” menu of the text editor.

For example, on left clicking the template ”Standard (with date)/multipage)” code is added to our (possibly empty) document. To save and compile the document select ”Compile”. Then, the page layout as depicted in Figure 2.1 is displayed.

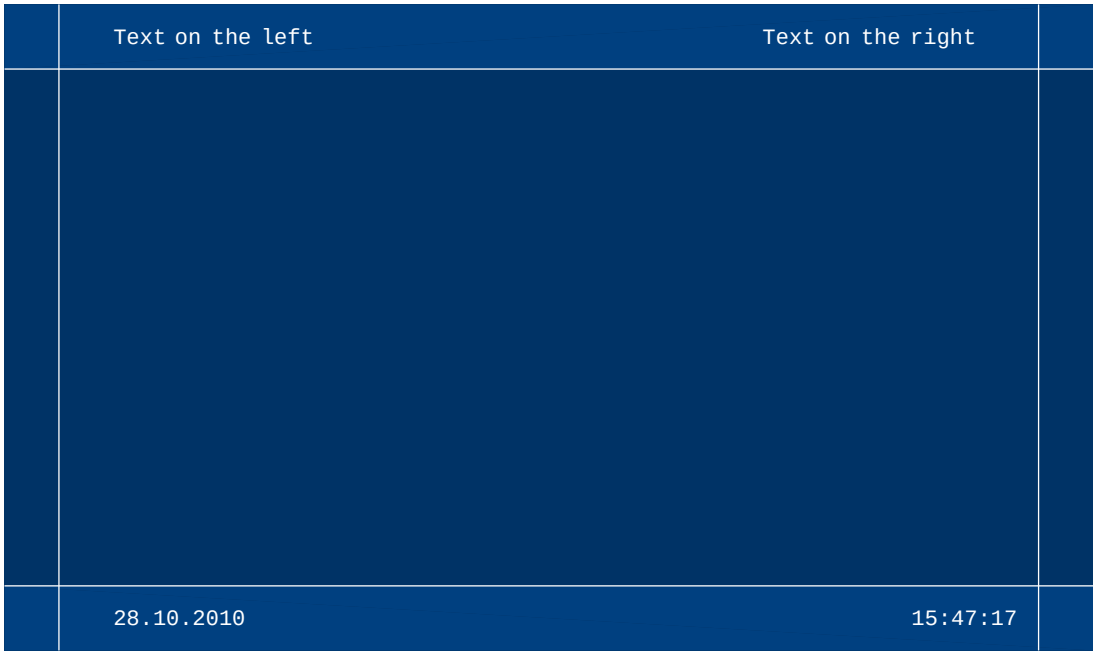


Fig. 2.1: The page layout template ”Standard (with date)/multipage”

Let us have a look at the code lines that were added to our document:

```
\Include(ProgDir + "\Shared\GUI\StdPageStyle.txt");

\begin{document}

\begin{#HeadLeft} text on the left \end#
\begin{#HeadRight} text on the right \end#
\Include(ProgDir + "\Shared\GUI\StdHeadFoot.txt");
\Std_BG();

\end{document}
```

We already learned the meaning of the `Include` commands in the first and fifth line. The inserted files contain page layout commands. Exchanging "text on the left" and "text on the right" by new text, it will be written in the top line on the left and on the right hand side, respectively. Finally, the command `\Std_Bg()` generates the white frame and color layout of the page.

Other page layout templates available in the templates menu modify the look of the frame and add the possibility insert footnotes. The last template in the list "simple (with preamble)" just defines a page with default background color and without frame. This template does not include further SigMath documents, but writes all necessary commands to alter background or text color, and positioning of the text directly to the preamble of the document instead.

In order to customize the page format we have to change the values of the three properties `LeftMargin`, `TopMargin`, and `TextWidth`.

Syntax:

```
\LeftMargin
\TopMargin
\TextWidth
```

All three commands can be placed anywhere in the code, but `TextWidth` is only treated during compilation. The properties take values between `0` and `1` and give distances proportional to screen width (for `LeftMargin` and `TextWidth`) and screen height (for `TopMargin`), respectively. Thus, e.g., setting `LeftMargin` to `1` means that it equals screen width. We indicate the distances that are controlled by the above properties by arrows in Figure 2.2.

2.3 Fonts

To change the font of text we use `selectfont`. Further properties of the selected font are adjusted by the properties `fontname`, `fontpoints`, and `fontcharset`.

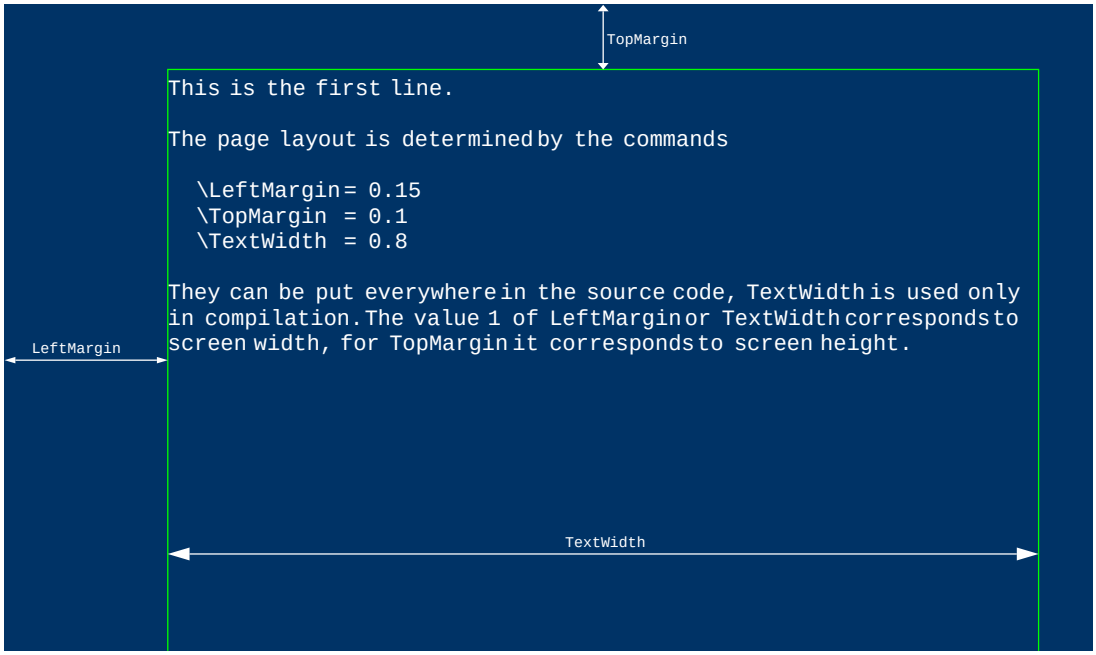

TopMargin
This is the first line.
The page layout is determined by the commands
\LeftMargin= 0.15
\TopMargin = 0.1
\TextWidth = 0.8
They can be put everywhere in the source code, TextWidth is used only
in compilation. The value 1 of LeftMargin or TextWidth corresponds to
screen width, for TopMargin it corresponds to screen height.
LeftMargin
TextWidth

Fig. 2.2: Effect of the page format parameters

Syntax:

```
\fontname{fontFamily}
\fontpoints{points}
\fontcharset{charSet}
\selectfont
```

The syntax corresponds to the commands in LaTeX. The argument `fontFamily` is the name of the font, i.e., e.g., `Arial` for normal text, `Courier new` for `monospaced` text. Monospaced text is typewriter-like, i.e., every letter has the same width (e.g., i and M). Available font families depend on the operator system on your computer. See the help pages of the system for more information.

The default font size in SigMath is 36 points. For the displayed text, this has to multiplied by the factor (*screen height in pixels*)/768. To change font size, assign the new font size (measured in points) to the property `fontpoints`. Typically, one is rather interested to rescale font size relative to the default size. This can be done by using one of the commands listed in Table 2.2. They can be used in the source code, directly. All text, following those commands then appears in the selected font size until font size is modified, again or a new environment starts (for environments, see Setion 2.1.

The property `fontcharset` can take the following values: `default`, `ansi`, `symbol`, `chinese`, `arabic`, `greek`, and `russian`. This property allows to load character sets used in the corresponding language as long as the operator system on your computer supports them.

Command	Factor	Sample
\tiny	0.33	tiny
\scriptsize	0.50	very small
\footnotesize	0.66	foot note
\small	0.75	small
\normalsize	1.00	normal
\large	1.33	large
\Large	1.50	larger
\LARGE	1.75	even larger
\huge	2.00	huge
\Huge	2.33	gigantic

Tbl. 2.2: Font scaling commands

The new values of the properties `fontname`, `fontpoints`, and `fontcharset`, only take action after the command `selectfont`. All modifications apply to the text region, only and only within the (rest of the) environment where they are specified.

Font shape can be changed by the following commands: To produce **bold face** write (\bf), for *italics* one uses (\it). To underline or overline text, use (\underline) and (\overline).

Syntax:

```
\it
\bf
\underline{text}
\overline{text}
```

Using the command without the curly brackets the modification applies to the whole text in the environment following the command. With brackets only the presentation of the `text` in `{text}` is changed.

2.4 Formatting

This section deals with formatting text. We already learned how to change font properties. Now, we want to consider alignment and spacing of text.

Text alignment

By default text is aligned left in SigMath. To alter this behavior we can set the property `FlushRight` to `true` to add right alignment to the text which results in justified text. The default behavior is restored by setting the property to `false`.

Syntax:
```
\FlushRight(true);
\FlushRight(false);
```

To center a certain part of the text we put it in a `centerline` environment.

Syntax:
```
\begin{centerline}
 centered text
\end{centerline}
```

A possible application of the `centerline` environment looks like this:

```
left aligned text
\begin{centerline}
 centered text\\
 on \\
 several lines
\end{centerline}
back to left aligned text
```

Line breaks

Line breaks are performed automatically in SigMath, without using hyphenation of words at the end of the line. To force a line break we can use one of the commands `linebreak`, `\\` or `newline`. Using `linebreak` the text in the line is displayed justified where spacing between the words is inserted if necessary. The commands `\\` and `newline` produce lines with normal spacing and left aligned text. The last two commands are in fact identical.

Syntax:
```
\linebreak
\\
\newline
```

Since SigMath's display is not restricted to a (limited) sheet of paper but on a basically infinite paper roll (by scrolling the screen) there is no page break in the usual sense. Instead, we can add links to new pages. Links will be treated in Section 3.1 in Chapter 3.

Spacing

If we want to modify the line spacing we set the property `BaseLineStretch` to a factor, that shrinks the line space (<1) or enlargens it (>1).

Syntax: `\BaseLineStretch`

For example, using

```
\BaseLineStretch = 2;
```

we double the line spacing.

Furthermore, the commands `VSpace` and `HSpace` insert vertical and horizontal space, respectively. As arguments we specify the spacing in terms of screen height or width.

Syntax:
```
\VSpace(vertDistance);
\HSpace(horDistance);
```

The arguments `vertDistance` and `horDistance` are of type `Real`. Using

```
word1 \HSpace(0.05); word2
```

we generate a horizontal spacing of 5% of screen width between the two words. Analogously we can add vertical spacing between, e.g., two paragraphs by using `VSpace`. Often used horziontal spacings can be inserted by special commands which are contained in Table 2.3. The factors are given are with respect to the height of the current font, providing comparable behaviour for all font sizes. The negative factor $-3/18$ corresponds to a shortening of the *horizontal* spacing to the next word by 3/18 of the *height* of the current font.

Command	**Description/ Factor**
\~	Space
\	Space
\,	3/18
\:	4/18
\;	5/18
\!	-3/18
\quad	1
\qquad	2

Tbl. 2.3: Shorthands for horizontal spacing

2.5 Positioning of text

If we want to place text at a precise location on the screen we have to perform two steps. First, we have to specify the position of the text block and then display the text at this position.

For the first task we can use the following commands:

Syntax:
```
\TextPos(targetX,targetY,textX,textY);
\TextPos(p,              textX,textY);

\TextPosPixel(targetX,targetY,textX,textY);
\TextPosPixel(p,              textX,textY);
```

The effect of those commands is as follows: First, a target point is specified that defines the origin for the positioning of the text block. This is given the arguments `targetX` and `targetY` or by the first two entries of the floating point variable p (see Chapter 7). The coordinates used in `TextPos` are relative screen coordinates, i.e., (0,0) denotes the bottom left corner and (1,1) the top right corner of the screen. `TextPosPixel` uses pixel coordinates, i.e., (0,0) denotes the bottom left corner and (`anim.Width-1,anim.Height-1`) the top right corner. Here, `anim.Width` and `anim.Height` give the number of pixels in horizontal and vertical direction, respectively (see Section 6.1). The remaining two arguments `textX` and `textY` determine how (in relation to the previously defined origin) the text block is aligned. Let us assume for the moment that the textblock consists of a single line, only. Virtually, we draw a rectangle around the text. Its width equals the width of the text block and its height is given by the font size that is valid in front of (!) the current text block. In local coordinates, the center of the rectangle has coordinates (0,0), the lower left corner is specified by (-1,1), and the upper right corner is given by (1,1). Now, `textX` and `textY` specify which point (in local coordinates) is mapped to the origin. For text blocks containing more than one line, the defining rectangle is only adjusted to the first line.

Finally, the content of the text block is given by a `postext` environment. There are two ways to specify a `postext` environment:

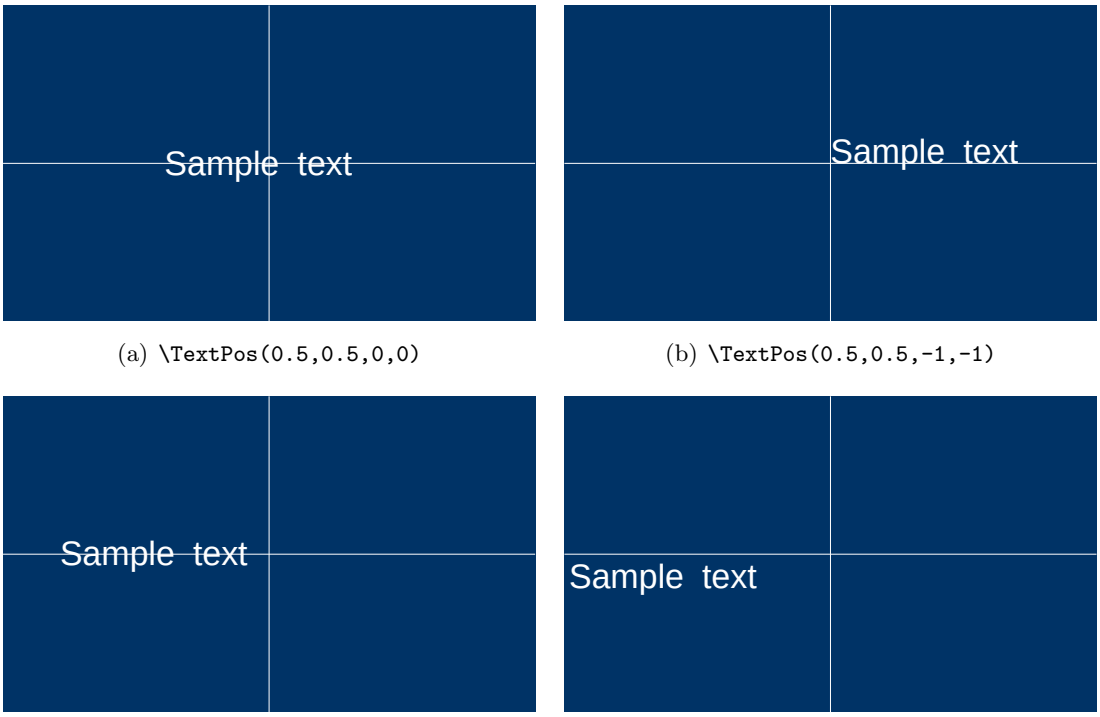
Syntax:
```
\postext{ % here you put text, commands }

\begin{postext}
 text or commands
\end{postext}
```

Example 2.1 Let us study the effect of text positioning on the text block "Sample text". The center of the screen is our target point and we use the following text coordinates

1. `\TextPos(0.5,0.5,0,0);`
2. `\TextPos(0.5,0.5,-1,-1);`
3. `\TextPos(0.5,0.5,1,0);`

4. `\TextPos(0.5,0.5,1.5,2);`

Note the placement of the descender of the letter p, especially in the second example. The results are depicted in Figure 2.3.



(a) `\TextPos(0.5,0.5,0,0)` (b) `\TextPos(0.5,0.5,-1,-1)`

(c) `\TextPos(0.5,0.5,1,0)` (d) `\TextPos(0.5,0.5,1.5,2)`

Fig. 2.3: Positioning of text using `TextPos`

For the positioning of text within plot environments we refer to Section 4.1.6, p. 63.

Though the text on the screen is displayed in two dimensions, there exists a third coordinate called ”depth”. It can be adjusted by changing the value of the property `TextDepth`.

Syntax: `\TextDepth`

The default value is `0`. Depth is used if SigMath has to decide which of two overlapping objects appears closer to observer and hence hides the other. Here, ”closer” means that the depth of the object in the foreground is lower than the depth of the object in the background. If `TextDepth` exceeds [-1000,1000] the text is not shown at all.

2.6 Enumerations

Enumerations are numbered listings used to sort text. They can be generated by using the `enumerate` environment. Nesting enumerations is possible. List items are defined by using the `item` command. To split longer text into several lines one has to use `\\` command. Following

lines, belonging to the same item are indented as the first line. To generate subitems we start a new (sub)list by the command `nextlevel` followed by `item` commands. To end the sublist, we write `prevlevel` and resume to the previous enumeration level. Now, new items can be added. The syntax for an enumerations thus looks as follows:

Syntax:

```
\begin{enumerate}
 \item item 1 of level 0
 \item item 2 of level 0\\
 second line of the second item of level 0
 \nextlevel
   \item level 1
   \nextlevel
     \item level 2\\
     second line of the last item in levelö 2
     \prevlevel
   \item level1
   \prevlevel
 \item item 3 of level 0
\end{enumerate}
```

2.7 Inserting figures

As we will see in forthcoming chapters SigMath offers multiple possibilities to generate (animated) figures on its own. Nevertheless, already existing figures can be embedded into a SigMath document. The command for inserting figures is the `Image` command. It generates a rectangle of user specified height and width and screen position that contains the figure. The `Image` command then loads the file and transforms its content to an object (a texture) which can be displayed by SigMath. If the figure does not change from frame to frame, it is not reloaded. Nevertheless, SigMath checkes this condition at any frame and loads a new figure if necessary (see the online help for an example). Furthermore, one can control how the figure is placed within its rectangle: display it using its own width and height, or (maybe) distort it to fill the whole rectangle. The corresponding behavior is adjusted by the `ImageStretch` property. Its value takes effect on the following `Image` commands in the document.

Syntax:

```
\ImageStretch = FILL;
\ImageStretch = PROP;
\ImageStretch = PROP_MAX_2;
\Image(name, [left, top,] width, height, depth);
```

On `FILL` the figure fills the whole rectangle and might be distorted; `PROP` rescales the figure preserving aspect ration to fit into the rectangle (default setting). Using `PROP_MAX_2` the scal-

ing is only performed up to a scaling factor of 2. The first argument `name` of `Image` specifies the image to be displayed. The image can be made available in different ways: `name` gives the filename (`*.bmp` or `*.jpg`) of the file that contains the image; `name` is ia text variable (see the paragraph on `String` in Section 6.2.3) that contains the filename; `name` is the name of an array (see Section 6.3.2); `name` is the name of a variable of type `Bitmap` (see Section 6.3.5). For arrays of numbers, every element is interpreted as a grey scale value and thus, a gray-scale image of the size of the array is displayed. The optional arguments `left` and `top` give the coordinates of the top left corner of the rectangle relative to screen coordinates (0 for left or bottom, resp.). The parameters `width` and `height` specify width and height of the rectangle relative to screen size and `depth` sets the depth of the rectangle. This variable is needed to detect foreground and background if objects overlap. Here, an object is closer to the observer if its depth is smaller. By default, `depth` is 0.

We give some examples to illustrate the effect of the different values of `ImageStretch`.

Example 2.2 We start using no optional arguments. That means, the figure is displayed at that position in the text at which it appears in code. The figure fills the whole rectangle, measuring a fifth of screen width in width and half of the screen height.

```
\begin{document}
\ImageStretch = FILL;
\Image(Path + "Image.jpg",0.2,0.5,0);
\end{document}
```

Example 2.3 To place the figure at a certain position in the rectangle we write the following.

```
\begin{document}
\ImageStretch = PROP;
\Image(Path + "Image.jpg",0,1,0.2,0.5,0);
\end{document}
```

Example 2.4 Finally, the rectangle is the whole screen. If the original figure is small, enlarging it by a large factor usually produces poor quality and graining. Thus, in this caes we bound the maximal scaling to 2.

```
\begin{document}
\ImageStretch = PROP_MAX_2;
\Image(Path + "Image.jpg",0,1,1,1,0);
\end{document}
```

2.8 Colors and background

For presentations on screen color is a fundamental tool. Thus, we explain the general principles of color usage in SigMath at this point. SigMath uses the RGB color model. Therby, RGB stands for the colors red, green, and blue. The RGB model is an additive color model, where the three basic colors add to white (optical mixing). A color is specified by three numbers: its red, green and blue component. Every component varies between 0 and 1. Thus all commands concerning colors have this three color components in the prescribed order as arguments. For example, `(1,1,1)` codes white and `(0,0,0)` codes black, `(1,0,0)` returns red, `(0,1,0)` green, and `(0,0,1)` blue. Variation of the color components yields all colors which can be displayed. Additionally, one can provide the alpha-value as a fourth component. By default its value is 1. It codes opaqueness of a given object and takes effect if several objects overlap. Using an alpha-value of 1 means opaqueness or 0% transparency. Image we look at a red ball behind a green-colored pane. If the pane has an alpha value of 1, the red ball behind is not visible. The lower the alpha-value of the window pane, the more of the ball is shining through. Finally, an alpha-value of 0 results in a transparent window pane.

With this basic principle in mind we can color different objects using one of the commands listed in Table 2.4 followed by the three color components as arguments.

Syntax: *ColorCommand*`(r,g,b,alpha);`

For example, using

```
\Color(1,1,1);
```

text color is set to white.

Every command that sets a color is accompanied by a command to return the current color of an object. The commands are simply given by adding the prefix `Get` to the command.

Syntax: `\Get`*ColorCommand*

For background design two further commands are available. `BgImage` allows to use an image as a wallpaper. The command behaves as the `Image` command in Section 2.7. As for the `Image` command the positioning and scaling of images is controlled by the `ImageStretch` property. The second command `BgColoredRect` produces smooth color gradients starting from specified colors in the corners of the screen.

Syntax: `\BgColoredRect(bottomleft,bottomright,topleft,topright);`

In this command, the colors of the corners are specified anti-clockwise starting at the bottom left corner using their RBG values and alpha-value. That means, the argument of `BGColoredRect`

Command	Description
`Color` `GetColor`	text color
`BgColor` `GetBgColor`	background color
`GoToColor` `GetGoToColor`	color of text in links
`InputColor` `GetInputColor`	color of text in inactive input fields
`InputColorFrame` `GetInputColorFrame`	color of the dashed frame around active input field
`InputColorSelText` `GetInputColorSelText`	color of selected text in active input fields
`InputColorSelBg` `GetInputColorSelBg`	text background color of selected text in active input fields

Tbl. 2.4: Commands for colors

consist of a comma-separated list of 16 entries with values between 0 and 1.

Example 2.5 To generate a color gradient as depicted in Figure 2.4 use the following code:

```
\BgColoredRect(0,0,0,0,
               1,0,0,0,
               0,1,0,0,
               0,0,1,0);
```

2.9 Tables

Tables can be generated in SigMath by using an `array` environment. Please, do not confuse the `array` environment with the complex data type `array` which will be introduced in Section 6.3.2. All commands concerned with the display of such variables will be introduced there.

Basically speaking, a table is just a special arrangement of ordinary text and thus all previously mentioned text layouting commands and properties like font family and font size carry through. As known from LaTeX the content of the table is written sequentially in the code. To place text in the next column one enters `&` and a new line is started by `\\`. If we want to add a horizontal line we put a `hline` before the next line break.

Syntax: `\hline`

One difference to the usage in LaTeX is the following: in SigMath the number of columns does

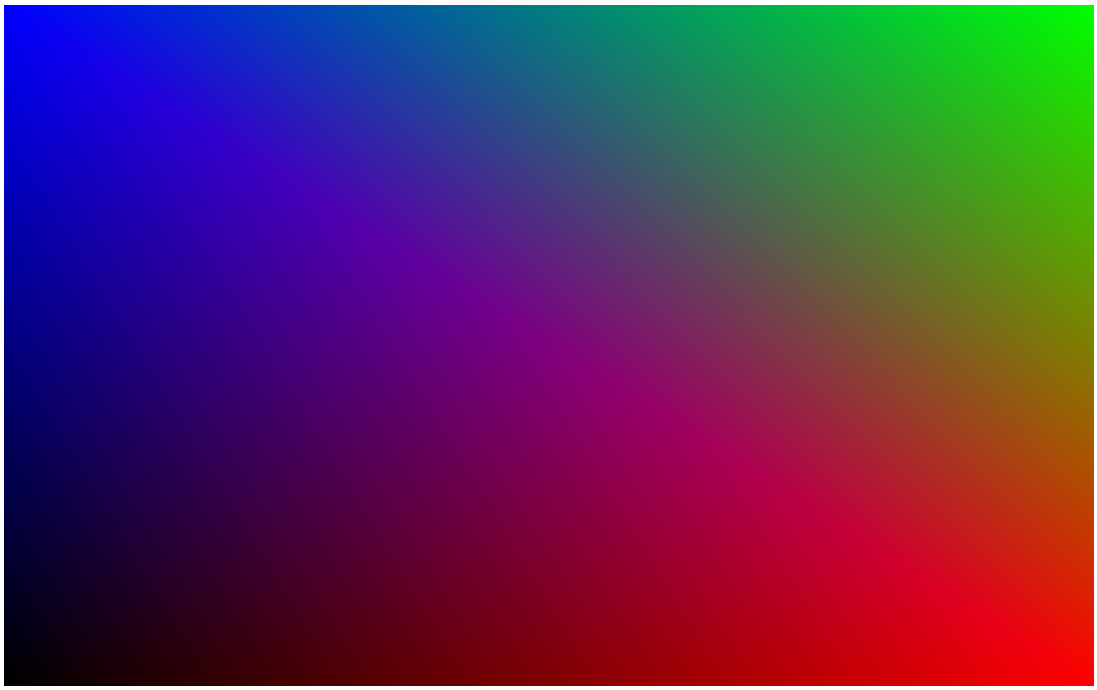


Fig. 2.4: Example for a color gradient

not have to be specified beforehand. That means, the token & acts as a tabulator in ordinary word processing sofware. Where the next tabulator is set depends on the longest width of the text in the same column. There is no automatic line break within a column and every row can be ended at any time, since no predefined number of columns has to be filled.

Spacing

Using the properties `ArrayColSep` and `ArrayStretch` the column separation distance and line space can be adjusted, respectively.

Syntax:
`\ArrayColSep`
`\ArrayStretch`

Their value is `1.0` by default. The column separation distance is given by `ArrayColSep` multiplied by the width of a blank in the current font size. The line space is given by `ArrayStretch` times the current font height. To get the current value of the column speration distance we can use the `ColSepDelta` command.

Syntax: `\ColSepDelta`

This command is read-only, i.e., we cannot alter the value of the separation distance in this way. Finally, all distance modifications are active for the current table, i.e., for the whole `array`

environment they are contained in.

Positioning in tables

Text alignment in columns is controlled by the `ColAlign` property.

Syntax:
```
\ColAlign(pos);
\ColAlign(i,pos);
```

In both cases `pos` is a `Real`-number. Default value is `1.0`, corresponding to right-aligned text. Left alignment is given by `0.0` and centering by `0.5`. Other values are admissible, especially, numbers larger than `1.0` or smaller than `0.0`. The second alternative changes the property only for the i-th column, where enumeration starts with `1`. The horizontal positioning of entries is given by the formula

```
beginning of column + pos × ( column width - width of entry).
```

Thus, the parameter `pos` does not take effect on those entries of the column that determine the width of the column. Since there are no restrictions on the number of columns or on their width, tables with few columns might appear "thin". To widen them it add horizontal spacing by writing, e.g., $\sim$ or `\quad`. This increases text width and, thus, increases column width.

Displaying matrices and vectors

To display the values of `array` data types on the screen, we use the command `ArrayBlock` (see p. 97 in Section 6.3.2 for further description and examples). In this way, several arrays can be written in block notation. Typically, matrices and vectors are written in brackets. SigMath uses the command `Brackets` to perform this task. Experienced SigMath users may customize this behavior by modifying the definition in the subdirectory `Shared/OpenGL/Array` of the program directory. In any case, the following line has to be added in the preamble to introduce the `Brackets` command to SigMath:

```
\Execute(ProgDir + "\Shared\OpenGL\Array\Brackets.txt");
```

Then `Brackets` is available. Just place it at the end of the `array` environment that contains the array.

Syntax: `\Brackets();`

Changing the entries of matrices and vectors on the screen

Using the `InputArrayBlock` command we can combine an `array` variable with the screen display in such a way that the user can change the value of the variable by direct input. More on this topic can be found in Section 3.3.

Determining the position of a cell

SigMath does not provide further commands to modify the layout of a table, directly. However, experienced programmers may use OpenGL for this purpose. One typical task is to put a frame around certain rows or columns in a table. In order to specify all necessary lines one has to determine the position of the text cells. For this purpose, SigMath provides the commands `ArrayRect`, `ArrayColX`, and `ArrayRowY`. The position of the currently selected entry is given by `ArrayGetCell`. The last command requires the index of the array, which is obtained by `ArrayIndex` (analogous to `PlotIndex`, see Section 4.1.3).

Syntax:

```
\ArrayRect(coord);
\ArrayRect(row1,col1,row2,col2,leftX,leftY,rightX,rightY);
\ArrayColX(coord);
\ArrayRowY(coord);
\ArrayIndex;
\ArrayGetCell(x,y,row,col,arrayInd);
```

The command `ArrayRect` returns in both versions the coordinates of a rectangle that encloses the specified entries in the table. The first version returns a four-dimensional vector, where the first component contains the horizontal screen coordinate of the first entry of the first column, the second component contains the vertical screen coordinate of this entry, and the third and fourth component contain the corresponding coordinates of the last entry in the last column. Using this coordinates we can draw a rectangle enclosing the whole array. If we only want to enclose certain elements of the array we use the second version. This requires as arguments the row and column indices of the top left and bottom right element. Keep in mind, that the first row or column has index `0`. Using this coordinates we can draw a rectangle with OpenGL commands. The following commands have to placed within the same `array` environment as the table. The rectangle is then drawn using green color.

```
ArrayRect(row1,col1,row2,col2,leftX,leftY,rightX,rightY);
gl.PolygonMode(gl.FRONT_AND_BACK,gl.LINE);
gl.Color(0,1,0);
gl.Rect(leftX,leftY,rightX,rightY);
```

Both commands `ArrayColX` and `ArrayRowY` require the argument `coord` consisting of the horizontal pixel coordinates of the positions where the columns start and the vertical pixel coordinates of the position where the rows start. Here, `coord` is a vector, whose size is determined by the number of columns or rows, automatically.

To ask for the value of certain elements of the array, we need to have access to a certain array. This is done by referencing to the *index* of an array. We suggest to save it to a variable of type `Integer` within the corresponding `array` environment.

```
\begin{array}
 entries of the array
 \arrayInd = ArrayIndex;
\end{array}
```

If we want to detect whether there is an entry of the `array` enviroment with index `arrayInd` at a point having pixel coordinates `(x,y)` we use the `ArrayGetCell` command. It returns `true` if the condition is fulfilled and writes column and row index to the variables `row` and `col`. If there is no entry at this point, `row` and `col` remain unchanged, and `ArrayGetCell` returns `false`. If there is no `array` environment with index `arrayInd` at all, an error message is returned.

Since the draw functions in the example are OpenGL commands, we postpone their description to the corresponding Chapter 7.

2.10 Symbols

Known to LaTeX users, SigMath provides several commands to insert frequently used symbols in the text. They are given in the following Table 2.5.

Greek lower case letters

α	\alpha	λ	\lambda	τ	\tau
β	\beta	μ	\mu	υ	\upsilon
γ	\gamma	ν	\nu	ϕ	\phi
δ	\delta	ξ	\xi	χ	\chi
ϵ	\epsilon	o	\o	ψ	\psi
ζ	\zeta	π	\pi	ω	\omega
η	\eta	ρ	\rho	ϑ	\vartheta
θ	\theta	ς	\varsigma	φ	\varphi
ι	\iota	σ	\sigma	ϖ	\varpi
κ	\kappa				

Greek uppercase letters

Γ	\Gamma	Δ	\Delta	Θ	\Theta
Λ	\Lambda	Ξ	\Xi	Π	\Pi
Σ	\Sigma	Υ	\Upsilon	Φ	\Phi
Ψ	\Psi	Ω	\Omega		

Mathematical symbols

$\forall$	\forall	$\exists$	\exists	$*$	\ast
$\cong$	\cong	$\perp$	\perp	$\sim$	\sim
$\leq$	\leq	∞	\infty	$\geq$	\geq

$\propto$	\propto	∂	\partial	$\bullet$	\bullet
$\neq$	\neq	$\equiv$	\equiv	$\approx$	\approx
$\otimes$	\otimes	$\oplus$	\oplus	$\emptyset$	\emptyset
$\cap$	\cap	$\cup$	\cup	$\supset$	\supset
$\supseteq$	\supseteq	$\not\subset$	\notsubset	$\subset$	\subset
$\subseteq$	\subseteq	$\in$	\in	$\notin$	\notin
$\angle$	\angle	∇	\nabla	$\prod$	\prod
$\surd$	\surd	$\cdot$	\cdot	$\wedge$	\wedge
$\vee$	\vee	$\sum$	\sum	$\int$	\int
$\pm$	\pm	$\times$	\times	$\div$	\div
$\neg$	\neg	$\lceil$	\lceil	$\lfloor$	\lfloor
$\rceil$	\rceil	$\rfloor$	\rfloor		

Multilingual support

č	\vc	Č	\vC	ů	\ou
ě	\ve	Ě	\vE	Ů	\oU
ř	\vr	Ř	\vR		
š	\vs	Š	\vS		

Miscellaneous symbols

€	\euro	...	\ldots	$\aleph$	\aleph
$\Im$	\Im	$\Re$	\Re	$\wp$	\wp
\	\backslash	$\sim$	\tilde	%	\%
{	\{	}	\}	$	\$
#	\#	&	\&		

Tbl. 2.5: Symbols in SigMath

3 User interaction

User interaction is an essential part of SigMath and extends its functionality and, especially, is flexibility a lot. Since programming interactive elements is comparably simple and intuitive, one can easily add interaction to word processing and programming task. Especially, all of SigMath's functionality around generating animations benefits from the possibility to interact with the user.

3.1 Hyperlinks

Hyperlinks or links are elements on a SigMath document that are connected to an action. The action is executed (every time) the user selects the link by clicking on the element. Typically, links refer to different pages or documents (as links in HTML documents) or execute certain functions.

Links to pages

The simplest form of an interactive element is a link to a (different) page. A page in SigMath is the displayed content of a SigMath document we can navigate through by using arrow keys or scrolling with the mouse. Pages may contain text and graphical objects whose generating commands are contained in the document block of a text file or are embedded by using `Include` commands (see Section 1.6.3). To generate new text or code that should be displayed on another page, we separate both pages by using the `newpage` command. This command can be interpreted as the SigMath-analog of "page breaks".

Syntax: `\newpage`

With this command a new page is generated within the current document that contains all content following `newpage` in the source. To make the page accessible from other pages, we have to assign page labels by using `label`.

Syntax: `\label{PageLabel};`

Labels can consist of letters and numbers. After that, one can add a link to the page by writing `GoToPage` in the SigMath document.

Syntax: `\GoToPage("text","label"[,file]);`

Here `text` denotes the text in the document, that is displayed. To emphasize its special meaning it is displayed in (a different) color (green by default). The color can be changed by using `GoToColor` (see Section 2.8 and Table 2.4). The second argument `label` contains the label of the target page. If `label` is empty, i.e., `""`, the link points to the first page of the current document. If the target is contained in another SigMath-document, one has to provide its filename in the last argument `file`. This argument is optional, if the links direct to a page contained in the same text document. For the distribution of pages to several documents one should have the following in mind: Opening a document in SigMath all code on all pages is compiled (i.e., transformed to a displayable SigMath document) even if they are (at first) invisible to the user. The more pages a document contains the more memory it will need. However, one has to take care that all variables and functions needed in the page are visible, that means valid (see Section 6.2.2). Thus, one has to decide which pages to be grouped in one document and which can (or should be) put into another document.

If one wants to use formatted text, for example containing special characters, in a `GoToString` command, one has to put an `inputlabel` environment in front of the `GoToString`:

Syntax:

```
\inputlabel{ here text and commands }

\begin{inputlabel}
 here text and commands
\end{inputlabel}
```

The first `GoToPage` or `DoString` command following this `inputlabel` environment uses the content of `inputlabel` instead of the value of `"text"`. Afterwards, the behavior of `GoToPage` returns to the default behavior.

By default, every pages has a number (by tracing link history) and a name (its label). Number and label can be read by using `CurrentPageNr` and `CurrentPageLabel`, respectively. In this way, we can link to a certain page.

Syntax:

```
\CurrentPageNr
\CurrentPageLabel
```

For example, using the command line

```
\CurrentPageNr = 2;
```

the second page in the history of the current text document is displayed. Be aware of the fact that this way of referencing pages is unreliable. Thus, we recommend the usage of labels.

Syntax: `\PageLabel(pageNr)`

The command `PageLabel` returns the label of the page with number `pageNr`. If no corresponding page exists, the empty String is returned.

Command links

To combine functions with elements in the user interface we use the `DoString` command. As with page links we can display colored text that executed a function if the user selects the text by left-clicking.

Syntax:
```
\DoString(text,fctName);
\DoString(text,fctName,ident);
```

The argument `text` is of type String and contains the text that should be displayed. It is highlighted by using the color `InputColor`. As with the `GoToPage` command one can display formatted text using an `inputlabel` environment. See page 40 for a description. The second argument `fctName` contains the name of the function, which is executed. The function must not have any arguments. It is given in quotation marks without parenthesis. The second version contains the argument `ident`, which can be used to pass the information to the function which `DoString` command called it. Without the third argument, `ident` is automatically set to `text`.

A small modification of the `DoString` command is given by the `DoValueOf` command.

Syntax:
```
\DoValueOf(varName,fctName);
\DoValueOf(varName,fctName,ident);
```

As before, clicking invokes a function. It displays the value of the variable `varName` (also compare the behavior of `ValueOf` in Section 6.2.2).

As was mentioned before, an identifier `ident` is given (implicitly or explicitely) which can be used to detect, what kind of link the function was invoked of. Thus, several different links can invoke the same function, executing different commands. In the function the value of `ident` can be requested by using `SelectedDo`.

Syntax: `\SelectedDo`

Example 3.1 As an example, we want to change displayed text of the link to the value of its identifier.

```
\variable S = "";
```

```
\function f() = { S = SelectedDo; }
\DoString("Text1","f");
\DoString("Text2","f");
Last function:  \ValueOf("S");
```

After compilation there is no text following the colon. Clicking, e.g., the link `Text1` the variable `Text1` appears after the colon. Clicking `Text2` afterwards, the displayed text changes to `Text2`, accordingly.

3.2 Input fields

Input fields provide a comfortable way to enter numbers or change them gradually. In particular, SigMath allows to modify the value of a parameter by pressing the right mouse button and moving the mouse up or down.

Input fields are generated by using the `Input` command.

Syntax: `\Input("nameOfVariable");`

The value that is entered or changed by the user is assigned to the variable `nameOfVariable`. To this end, the variable has to be defined and initialized, beforehand. We can specify the way of input, e.g., determine the number of decimal places displayed or if there are limitations on the value of the variable or step sizes. The number of decimal places is determined by the `Digits` property.

Syntax: `\Digits`

Here, 1 stands for one decimal place, 3 would be three decimal places and a negative value makes the number disappear. The acceptable ranges of values of `nameOfVariable` are determined by the three properties `InputMin`, `InputMax`, and `InputDelta`.

Syntax:
`\InputMin`
`\InputMax`
`\InputDelta`

`InputMin` and `InputMax` can attain arbitrary real numbers. If the user tries to enter a number smaller than `InputMin` or larger than `InputMax` the displayed input variable is automatically set to the appropriate bound. Mind, all variables have to be known to SigMath before using `Input` (see the paragraph on "Definition and initialization" in Section 6.2.2). Otherwise, modifications take effect on the following `Input` field.

As an example, suppose we only want to accept nonnegative entries. Then we can write

```
\InputMin = 0;
```

For unconstrained variables we use `INF` for infinity and `-INF` for negative infinity. An upper unbounded variable is thus given by

```
\InputMax = INF;
```

The third property specifies the precision and hence step sizes if the user moves the mouse up and down while pressing the right mouse button.

Example 3.2 The next program should allow the user to adjust the background color. Therefore, we need a variable of type `Real` for each color component red, green, and blue. By declaring the variables every variable is assigned a starting value, the variable is *initialized*. In other words, the variable `R` is assigned the `Real` constant `0.0`. Specifying `InputMin`, `InputMax`, and `InputDelta` monitor the input, which is realized by `Input`.

Code: (BackgroundColor.txt)

```
\variable R = 0.0;
\variable G = 0.0;
\variable B = 0.0;

\begin{document}
  \InputMax = 1; \InputMin = 0; \InputDelta = 0.01;
  R = \Input("R"); G = \Input("G"); B = \Input("B");
  \BGColor(R,G,B);
\end{document}
```

Finally, we can determine the last input field which was selected by the user by calling `FocusedInput`.

Syntax: `\FocusedInput`

This property is useful when one has to process WINDOWS messages, see Section 3.4 and Example 3.3.

3.3 Input vectors and input matrices

If we want to display and modify `array` variables we have to use `InputArrayBlock` instead of `Input`. For the moment, an `array` variable is just a way to store matrices or vectors in SigMath. A precise definition is given in Section 6.3.2. With `InputArrayBlock` we can allow or forbid to change entries of the matrix.

Syntax:

```
\InputArrayBlock(arrayName);
\InputArrayBlock(arrayName,mHandler);
```

The argument `arrayName` provides the name of a variable of type `array` of dimension 1 (vector) or 2 (matrix), in analogy to `NameOfVariable` in the `Input` command. The size of the vector or matrix is not specified beforehand and can depend on user input. The `array` environment which contains the `InputArrayBlock` command is refreshed in every frame. In the second version `mHandler` gives the name of a function that processes certain messages. The function must be of the following type:

```
\function mHandler(message,i,j) = {...}
```

Here, the `Integer` variables `i` and `j` give row and column indices of the current element and the variable `message` may take one of the values listed in Table 3.1. For vectors, `i=0`.

Value	Description
`M_IS_INPUT_ENTRY`	Before displaying an element of `InputArrayBlock` the message function given by `mHandler` is invoked with argument `M_IS_INPUT_ENTRY`. If `mHandler` returns `false` the value of the entry can not be modified by the user.
`M_BEFORE_DRAW`	Before displaying an element of `ArrayBlock` or `InputArrayBlock` the message function is called using the argument `M_BEFORE_DRAW`. If the function returns a variable of type String, instead of the element, the value of that variable is displayed.
`M_MOUSE_DOWN`	If the user left-clicks an element of the `InputArrayBlock`, the message function is called using the argument `M_MOUSE_DOWN`. If the function returns `true` SigMath ignores the message, i.e., the element will not be selected.
`M_MOUSE_UP`	Same action as in `M_MOUSE_DOWN`, but the message function is invoked after releasing the mouse button.
`M_BEFORE_CHANGE`	before an element is changed by user input, the message function is called using the argument `M_BEFORE_CHANGE`. This allows to, e.g., save the previous value of the element.
`M_AFTER_CHANGE`	After an element has been changed by user input, the message function is invoked using the argument `M_AFTER_CHANGE`. This can be used to reset an element to its previous value (saved by `M_BEFORE_CHANGE`).

Tbl. 3.1: Messages for input arrays

Further examples on message processing connected with matrix and vector inputs can be found in the online help.

3.4 Event handling

Another interactive part of SigMath is the ability to react to events automatically. Events may be user induced (e.g., the intention to close a document or change pages, or to use an input field) or might be connected to the operating system. Basically, events allow to interact on keyboard strokes or mouse movement. The following commands allow to connect the corresponding events to functions.

Syntax:

```
\OnDocClose(fctName);
\OnPageChange(fctName);
\OnVarChange(varName,fctName);
\OnWinMessage(fctName);
```

The argument `fctName` contains the name of the function that is called on the occurrence of the corresponding event. For example, if the user wants to close a document, the boolean function `fctName` is called if one uses `OnDocClose`. Depending on the return value of the function the document closes or remains opened. The command `OnPageChange` calls the function `fctName` which needs an argument of type Integer before switching to a new page. The value of the argument is the page number of the target page. As before, the page is only changed, if the function returns `true`. After the user changes the value of the variable `varName` the function `fctName` is called if one uses `OnVarChange`. In this case, the value of the variable is already changed since the next frame is already shown. The command `OnWinMessage` is the most flexible (and most complex) of the four commands. Thus, there are a lot of requirements on the function `fctName`. As an example we consider the function `ProcessMessage`.

Syntax:

```
\function ProcessMessage(message,arg1,arg2) = {
 commands
}
```

A function in `OnWinMessage` needs three arguments and returns a boolean value. If `true` is returned, the message is marked as done. The first argument `message` contains the name of the WINDOWS message, which can be one of the values contained in Table 3.2. Note that *system keys* are all key combinations containing the *Alt* key.

The arguments `arg1` and `arg2` have different meaning depending on the value of `message`. All messages concerned with keyboard events pass the ASCII code of the corresponding key as `arg1`. On mouse events, `arg1` and `arg2` contain the horizontal and vertical coordinates of the mouse cursor. ASCII code also plays a role in determining, whether a certain key is pressed.

Identifier	Description
WM_KEYDOWN	key pressed
WM_KEYUP	key released
WM_SYSKEYDOWN	system key pressed
WM_SYSKEYUP	system key released
WM_LBUTTONDOWN	left mouse button pressed
WM_RBUTTONDOWN	right mouse button pressed
WM_LBUTTONUP	left mouse button released
WM_RBUTTONUP	right mouse button released
WM_MOUSEMOVE	mouse movement

Tbl. 3.2: WINDOWS system messages

This can be checked by `IsKeyDown`.

Syntax: `\ui.IsKeyDown(code);`

It returns `true` if and only if the key having ASCII code `code` is pressed. ASCII code maps keys on the keyboard to numbers between 0 and 255. The first 32 characters are non-visible control characters, the numbers 0 to 9 have ASCII code 48 to 57, latin uppercase letters A to Z have code 65 to 90 and latin lowercase letters a to z correspond to 97 to 122. The assignment of the characters with higher ASCII code is system (and language) dependent. For special control characters we refer to Table 3.3.

Identifier	Description
VK_CONTROL	*CTRL*
VK_UP	*Arrow up*
VK_DOWN	*Arrow down*
VK_LEFT	*Arrow left*
VK_RIGHT	*Arrow right*
VK_ESCAPE	*ESC*
VK_PRIOR	*PageUp*
VK_NEXT	*PageDown*
VK_SHIFT	*Shift*
VK_SPACE	*Space*
VK_F1	function key *F1*

Tbl. 3.3: Functions returning ASCII code of special keys

The code of the other function keys *Fi* can be obtained by adding `i-1` to the value of `VK_F1`.

Example 3.3 Imagine we want to allow change of a value in an input field by arrow keys in addition to mouse action or keyboard input. Typically, we initialize the variable connected to the input field by a default value and provide information on maximal and minimal values and on the step size. Using the arrow keys the value of the variable shall be increased or decreased by the value of the step size, depending on whether we use *Arrow up* or *Arrow down*. To enable this behavior (for an activated input field) we have to add the following lines of code:

Code:

```
\variable X = 0.0;
\function ProcessMessage(message,arg1,arg2)=
{
 if (message == ui.WM_KEYUP)
 {
   switch(arg1)
   {
      case ui.VK_UP :
           X += InputDelta;
           return;
      case ui.VK_DOWN :
           X -= InputDelta;
           return;
   }
 }
}

\begin{document}
 \InputDelta=0.1;
 X=\Input("X");
 \OnWinMessage("ProcessMessage");
\end{document}
```

To get the position of the mouse cursor the following functions are available.

Syntax:

```
\ui.MouseX
\ui.MouseY
\ui.MouseDeltaX
\ui.MouseDeltaY
```

Here, `MouseX` and `MouseY` return the current position of the mouse cursor in relative screen coordinates. `MouseDeltaX` and `MouseDeltaY` can be used to determine the difference of the mouse position from the last frame to the current frame. These commands are read-only. To

set the position of the mouse cursor, we can use `SetMouseXY`.

Syntax: `\ui.SetMouseXY(x,y);`

Here `x` and `y` are numbers between 0 and 1 that specify the relative screen coordinates. The can be given by time-dependent functions, allowing the mouse to move independent of user action.

To determine whether a mouse button is pressed, we can use the commands `IsLeftButtonDown` and `IsRightButtonDown`.

Syntax:
```
\ui.IsLeftButtonDown
\ui.IsRightButtonDown
```

To customize the layout of the mouse cursor, we have to adjust the value of the property `Cursor` to one of the values contained in Table 3.4.

Syntax: `\ui.Cursor`

Value	Description
`crCross`	cross hairs
`crDrag`	pointing hand
`crHandPoint`	pointing hand
`crHourGlass`	hourglass
`crNo`	circle, crossed
`crSizeNESW`	diagonal double arrow
`crSizeNS`	vertical double arrow
`crSizeNWSE`	diagonal double arrow
`crSizeWE`	horizontal double arrow
`crUpArrow`	top arrow
`crArrow`	standard cursor

Tbl. 3.4: Mouse cursor

3.5 Draw and move selected objects

The following interactive function is a mouse interaction. We can draw objects and move them. The most important command for this is `SelObj`.

Syntax: `\SelObj(drawFct,messageFct,numSelObj);`

The function `drawFct` contains OpenGL commands to draw the objects (see chapter 7) and has no arguments. It is only called on certain mouse events. The calling event can be determined by `SelObjMessage` which is set automatically on calling the function `drawFct`.

Syntax: `\SelObjMessage`

Only the events `WM_LBUTTONDOWN`, `WM_LBUTTONUP` and `WM_MOUSEMOVE` are detected. By pressing the left mouse button a certain object is selected, releasing the mouse button deselects the object. If an object is selected it is passed a message if the mouse is moved. The function `messageFct` has 2 arguments of type Real where the first specifies the horizontal and the second the vertical coordinate of the mouse cursor. If `SelObj` is contained in a `plot2d` environment, the coordinates are with respect to the plot coordinate system. Otherwise, they denote relative screen coordinates. Using the argument `numSelObj` we can specify the number of objects to be generated. If we want to generate several identical objects we can use the same function. The objects differ by their identifier and are counted starting with 0 to `numSelObj-1`. On calling the functions `drawFct` and `messageFct` the identifier is assigned to the value of `SelObjInd`.

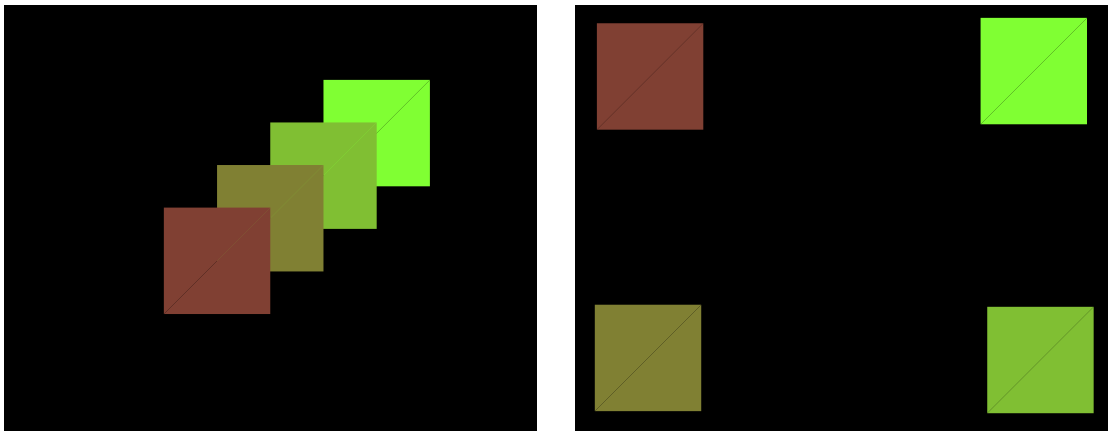
Example 3.4 We want to generate 4 slightly translated but overlapping squares which can be dragged using the mouse.

Code:

```
\variable X = RealArray(0);
\variable Y = RealArray(0);
\X.FillFromString("0.4 0.5 0.6 0.7");
\Y.FillFromString("0.4 0.5 0.6 0.7");
\X *= anim.Width;
\Y *= anim.Height;
\variable D = anim.Width*0.1;

\function Draw() =
{
 variable x = X(SelObjInd);
 variable y = Y(SelObjInd);
 gl.Color(0.5, (SelObjInd+1)/4.0, 0.2);
 gl.Translate(0, 0, 1);
 gl.Rect(x - D, y - D, x + D, y + D);
}

\function OnMessage(x,y) =
{
 switch(SelObjMessage)
```

```
 {
   case ui.WM_MOUSEMOVE:
        X(SelObjInd) = x*anim.Width;
        Y(SelObjInd) = y*anim.Height;
        return;
   case ui.WM_LBUTTONDOWN:
        Beep(-1);
        return;
   case ui.WM_LBUTTONUP:
        Beep(1);
        return;
 }
}

\begin{document}
 \SelObj("Draw", "OnMessage", 4);
\end{document}
```

First, four horizontal and vertical arguments are read and converted to screen pixels. The function `Draw` the lowercase `x` and `y` of the current object (given by `SelObjInd`) are read and squares with center `(x,y)` are drawn. The next function `OnMessage` which has 2 arguments, executes certain commands depending on the nature of the calling event. On pressing or releasing a key a sound is played, on mouse movement the position of the current object is changed. In The original view after compilation is depicted in Figure 3.1(a), and the result after mouse movement can be seen in Figure 3.1(b).



(a) original view (b) after motion

Fig. 3.1: Four squares and mouse motion

4 Plots in SigMath

This chapter deals with the visualization of mathematical objects like functions, vectors. We will learn to define axis and how to modify their properties, add or change background, and how to modify the resulting graphics, interactively. With this simple techniques we will be able to generate highly detailed graphics.

In order to tell SigMath's interpreter that the following commands are meant to generate a plot we enclose that code in `plot2d` and `plot3d` environments, respectively. A `plot2d` environment generates a plot which has an identifier (a number) for later reference. Within a plot, i.e., within a `plot2d` environment, we draw graphics using the `Plot` command. Every `Plot` command adds a plot item which, again, gets a number as identifier. Thus, every single plot item within a plot can be modified later on without having to redesign the whole plot, again.

4.1 Two-dimensional plots

4.1.1 Generating plots

We begin with the generation and modification of two-dimensional plots. Basically all we need to generate a plot is the `Plot` command. Depending on its calling arguments SigMath automatically chooses the appropriate way to visualize the object. Thus, `Plot` applies to functions and vectors in canonical or parameter representation. Every plot item generated from a `Plot` command has a number of properties that can be modified to change the look of the resulting plot, drastically. The syntax looks as follows:

Syntax:

```
\Plot(fctName,first,last[,xmin,xmax]);
\Plot(vectorName[,xmin,xmax]);
\Plot(fctNameX,firstX,lastX,fctNameY,firstY,lastY);
\Plot(vectorNameX,vectorNameY);
```

The first variant computes the graph of a previously defined function `fctName` on the interval `[first,last]`. If no further arguments are given, the function is drawn on that interval. Otherwise it is only drawn on `[xmin,xmax]`. The visualization uses equidistant sample points of the function in the interval. The number of points can be changed using the property `PlotNumPointsX2D`. Basically, in all variants the points are connected by lines (piecewise linear interpolation). Thus, if `PlotNumPointsX2D` is low, even smooth functions may look non-smooth.

In the second variant, we pass the name of a vector `vectorName` as the first argument. Its entries are drawn versus the discrete x-axis starting with 1 up to the length of the vector. If

optional arguments `xmin` and `xmax` are specified, the points are only drawn on the interval `[xmin,xmax]`. The analogy to the first variant is obvious, since evaluating a function at finitely many points just generates a vector, containing the function values.

The third and fourth variant use arguments in parameter representation. Here `fctNameX` denotes a function that is evaluated on the interval `[firstX,lastX]` and determines the horizontal coordinate of the image. Analogously, `fctNameY` determines the vertical coordinates. The displayed axis ranges are determined by the values of the functions on the intervals [first,last] (with respect to X and Y, resp.). In the fourth variant, the lengths of both vectors have to match.

After introducing the basic plot commands we want to add a plot to an existing document. As we already pointed out, all plot commands for a two-dimensional plot have to be put in a `plot2d` environment. Before SigMath can display the plot, we also have to specify a window which contains the plot. This is done using the properties `PlotWidth` for the width and `PlotHeight` for the height given in relative screen coordinates (1 stands for the whole screen width or height, 0.5 for half the screen. Minimum is 0.01 and maximum is 10). If we do not specify the position of the window explicitely, it will appear at the same position as it is written in the source code. Otherwise, we can specify its position using the property `PlotPos`. The top left corner of the frame is given in relative screen coordinates, i.e., e.g., (0,0) for the top left corner or (0.25, 0.5) for a fourth of screen width to the left and half the screen height down.

```
\begin{plot2d}
 ...
 \PlotWidth = 0.5;
 \PlotHeight = 0.5;
 \PlotPos(0.25, 0.25);
 \Plot(...);
 ...
\end{plot2d}
```

Example 4.1 Let us have a look at a simple example. We want to draw the graph of the sine function on the interval $[0, 2\pi]$.

Code: (Sine.txt)

```
\function f(x) = { return sin(x); }

\begin{document}
\BgColor(0.0, 0.2, 0.4);
\Color(1.0,1.0,1.0,1.0);
\begin{plot2d}
  \PlotWidth = 0.8;
  \PlotHeight = 0.8;
  \PlotPos(0.1,0.9);
  \Plot("f",0,2*Pi);
```

```
\end{plot2d}
\end{document}
```

The resulting SigMath output is depicted in Figure 4.1.

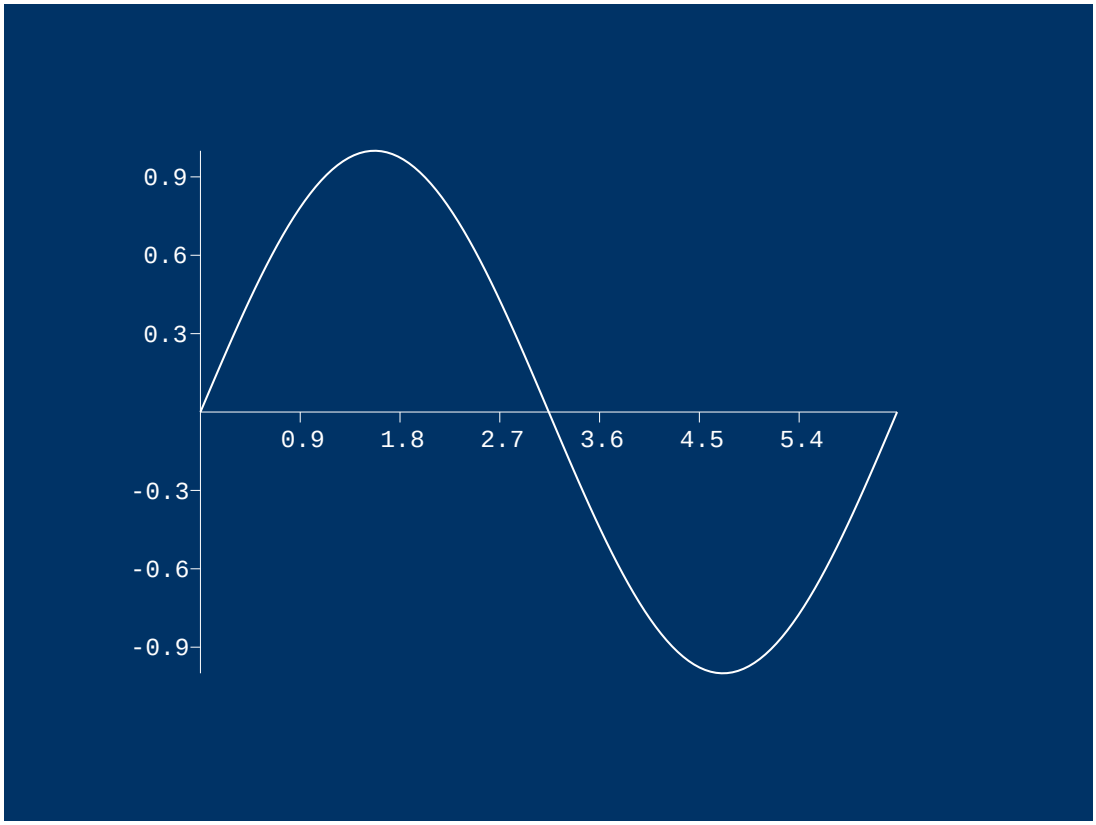
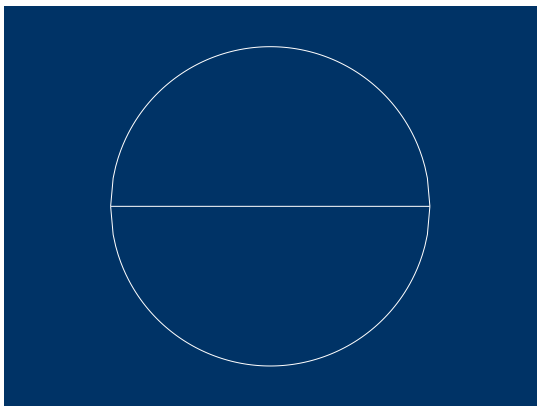


Fig. 4.1: The sine function on the interval $[0, 2\pi]$

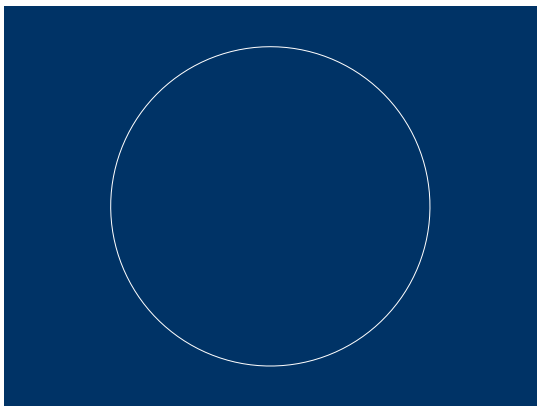
Example 4.2 The second example gives to alternative ways to display a circle. The first variant builds the circle from two semi-circles described by the functions $y_{1,2} = f(x) = \pm\sqrt{1 - x^2}$.

Code: (2Semicircles.txt)

```
\function f(x) = { return sqrt(1-x*x); }
\function g(x) = { return -sqrt(1-x*x); }

\begin{document}
\BgColor(0.0,0.2,0.4);
\begin{plot2d}
  \PlotBgColor(0.0,0.2,0.4);
  \PlotBgStyle = FILL;
  \PlotHeight = 1.0;
  \PlotWidth = anim.Height/anim.Width*PlotHeight;
  \PlotPos((1-anim.Height/anim.Width)/2,1.0);
  \PlotAxes = "1";
```



(a) Two semi-circles. Mind "kinks" at the left and at the right



(b) One circle in parameter representation

Fig. 4.2: Plotting a circle in two different ways.

```
  \AutoTicks = "";
  \PlotLineWidth = 3;
  \Plot("f",-1,1);
  \Plot("g",-1,1);
\end{plot2d}
\end{document}
```

Next, we want to draw the circle using its canonical parameter representation. As we know, the points on a circle fulfill the sine-cosine-law.

Code: (ParaKreis.txt)

```
\function f(x) = { return sin(x); }
\function g(x) = { return cos(x); }

\begin{document}
\BgColor(0.0,0.2,0.4);
\begin{plot2d}
  \PlotBgColor(0.0,0.2,0.4);
  \PlotBgStyle = FILL;
  \PlotHeight = 1.0;
  \PlotWidth = anim.Height/anim.Width*PlotHeight;
  \PlotPos((1-anim.Height/anim.Width)/2,1.0);
  \PlotAxes = "";
  \PlotLineWidth = 3;
  \Plot("f",0,2*Pi,"g",0,2*Pi);
\end{plot2d}
\end{document}
```

We have to adjust the width of the plot to make sure that the circle really looks like a circle.

Otherwise, the plot frame and hence the axis are scaled according to the screen dimensions. Typically, the screen is not square! We use relative commands that use screen dimensions provided by the operating system. Another interesting aspect is that the second variant looks "smoother", especially at the points, where the two semicircles join, see Figure 4.2. The reason is that in the first variant, sample points are uniformly distributed on the x-axis. Thus, the rightmost and leftmost neighbors are too far away to generate a properly smooth curve. To reduce this effect one has to increase the number of sample points. In the second variant, 128 points are uniformly distributed along the whole curve which results in a smoothing of the displayed curve.

In addition to creating a plot of an object the `Plot` command also returns the index of the generated plot item. This index can be stored in a variable and can be used later on to modify several properties of the plot. We continue this explanations at a later point.

4.1.2 Color, style, and background

Let us introduce some of the properties a plot has. All modification of these properties take only effect on the following `Plot` commands. In order to change the color of a graph, we assign to the `PlotColor` property the new color using its RGB coordinates (see Section 2.8). The line width used to visualize the graph of a function can be adjusted using `PlotLineWidth`.

Syntax: `\PlotLineWidth`

It can take any positive integer and is measured in pixels. Furthermore, the style of a graph can be modified. To this end, one can add one of the following lines to the source code:

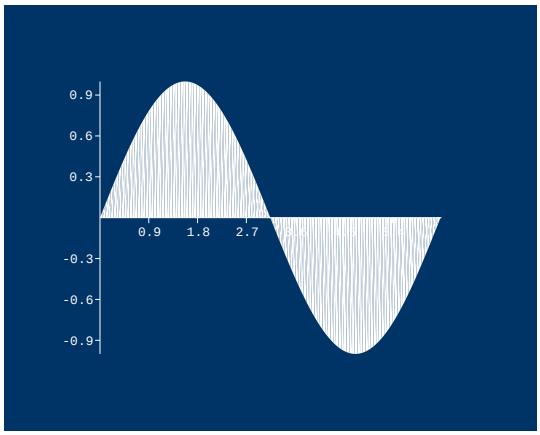
Syntax:

```
\PlotStyle = LINE_STRIP;
\PlotStyle = FILL;
\PlotStyle = RECT;
\PlotStyle = FILLED_RECT;
\PlotStyle = POINT;
```

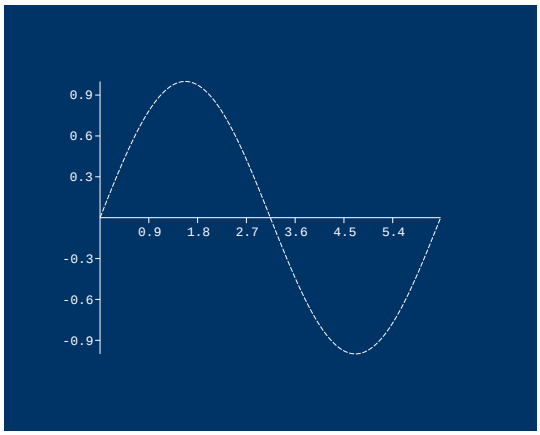
The option `LINE_STRIP` generates a line pattern using solid lines as default. `FILL` fills between the graph and the x-axis using grey color. The option `RECT` has typically the most impact if one uses only a very low number of sample points. The function is then approximated by step functions. `FILLED_RECT` combines the effect of `FILL` and `RECT`, i.e., it generates filled rectangles. Finally, `POINT` only draws the sample points without connecting them by straight lines. To change the line pattern used in `LINE_STRIP` one can modify the `PlotLineStipple` property.

Syntax: `\PlotLineStipple(lineStippleFactor, lineStipplePattern);`

The pattern is given as a `String` of 0 and 1's of length 16. Here, 1 stands for a drawn pixel and 0 for an "empty" pixel. Thus, the line pattern can be designed on a length of 16 pixels.



(a) `PlotStyle = FILL`



(b) `PlotLineStipple(1,"1111111111110000")`

Fig. 4.3: `PlotStyle` options

We can stretch this pattern using the `lineStippleFactor`. One typical pattern is given by `"0000000011111111"` which generates 8 empty pixels and 8 pixels of line. The same line would have been generated with lineStippleFactor 8 and `"0101010101010101 "`. If we want to return to the default behavior, we write `PlotLineStipple()` before using the next `Plot` command.

If we choose `POINT`, we can specify the size of the points in pixels. This is done by assigning a positive integer to `PlotPointSize`. For example, on `PlotPointSize = 3` the point is drawn as a square of size 3x3 pixels.

Syntax: `\PlotPointSize`

If several plots are contained on one page we can use the `PlotDepth` property to specify which objects are near to the observer and, hence, overlap the ones, which are far away. The lower the value of `PlotDepth` the closer the object to the observer.

Syntax: `\PlotDepth`

By default, the value is 0, i.e., if we want to add a plot that shall be shown in front of all other objects without specified depth we have to set `PlotDepth` to, e.g., -1.

As we already learned in the introduction, we can change the background of a plot as we have done it with the screen background. For this we need the two properties `PlotBgStyle` and `PlotBgColor`. The syntax is the following:

Syntax:

```
\PlotBgStyle = NONE;
\PlotBgStyle = RECT;
\PlotBgStyle = FILL;
```

```
\PlotBgColor(r,g,b);
```

The option `NONE` gives an empty background and no frame, `RECT` generates a frame around the plot window, and `FILL` allows to fill the whole plot window.

If we have chosen one of the last two properties we can adjust the color of the frame and the background using the property `PlotBgColor` in the usual RGB notation. For other background designs we can put a wallpaper in the background using the `Image` command (see Section 2.7). Make sure to specify an appropriate value for the depth of the image!

Basically the whole plot window is used to draw the plot. If we want to add margins we can specify its corresponding sizes by using `PlotMargins`.

Syntax:

```
\PlotMargins(left,right,top,bottom);
\PlotMargins(all);
```

All parameters are numbers which specify the size of the margin, using relative length with respect to the plot window. Using the first variant, the size of the four margins is specified, separately. The second variant treats all four margins the same. The default setting is given by `PlotMargins(0.1,0.1,0.1,0.1)` or, equivalently `PlotMargins(0.1)`, i.e., a tenth of the width and height is left for the margin.

4.1.3 Redrawing plots

Let us continue with the considerations on plot referencing. We already learned that every plot item has an identifier, its *index*. The index of a plot can be accessed within the corresponding `plot2d` environment using `PlotIndex`. To change properties of a plot or of a certain plot item, one should save the corresponding indices to variables. Then `BindPlot` and `BindPlotItem` can be used to select a plot or plot item in order to change some of its properties. Selecting a plot item the corresponding plot is *activated*.

Syntax:

```
\BindPlot(index);
\BindPlotItem(index);
```

After that command, all property of the plot can be modified. We can change the color of the background, adjust line patters or request axis areas of the active object (see Section 4.1.4). With `PlotReDraw` we force SigMath to redraw the plot using the new properties.

By default, a plot is redrawn automatically if one variable used in the depicted function is changed by user action. Also, if the version number of an array changes (see p. 95 in Section 6.3.2), the plot is redrawn automatically.

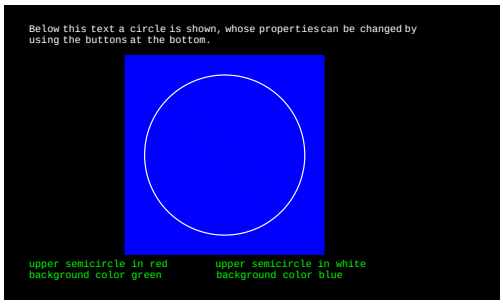
We provide an example to summarize this description.

Example 4.3 We want to generate and draw a simple, combined plot. Some objects shall be made adjustable by clicking several interactive elements. Compiling the file `BindPlot.txt` a white-colored circle on blue background is displayed (see Figure 4.4). In this example, the circle is generated using two semicircles (2 plot items). With the elements at the bottom of the screen we can change the color of the upper or lower semicircle, or background color. To change the color of one of the semicircles we have to activate the corresponding plot item. To change background color, the whole plot has to be activated.

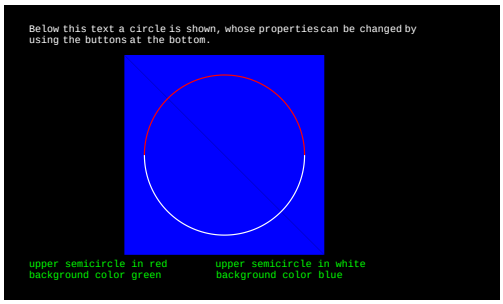
Code: (BindPlot.txt)

```
\function f(x) = { return sqrt(1-x*x); }
\function g(x) = { return -sqrt(1-x*x); }
\variable PI   = 0; % Plot index
\variable PII1 = 0; % PlotItem index
\variable PII2 = 0;
\variable size = 0.68;
\function UpperCircleRed() =
{
 BindPlotItem(PI1);
 PlotColor(1,0,0);
 PlotRedraw();
}
\function UpperCircleWhite() =
{
 BindPlotItem(PI1);
 PlotColor(1,1,1);
 PlotRedraw();
}

\function BgGreen() =
{
 BindPlot(PI);
 PlotBgColor(0,1,0);
 PlotRedraw();
}

\function BgBlue() =
{
 BindPlot(PI);
 PlotBgColor(0,0,1);
 PlotRedraw();
```

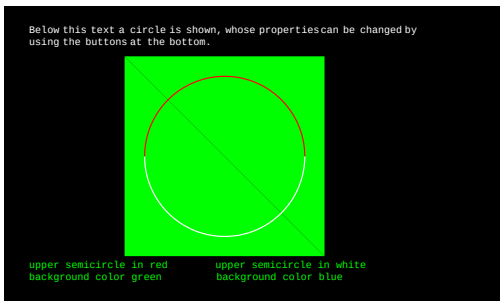
```
}

\begin{document}
Below this text a circle is shown, whose properties can be changed
using the buttons at the bottom.
\begin{centerline}
 \begin{plot2d}
   \PI = PlotIndex;
   \PlotWidth = size*anim.Width/anim.Height;
   \PlotHeight = size;
   \PlotAxes = "";
   \PlotBgStyle = FILL;
   \PlotBgColor(0,0,1);
   \PI1 = Plot("f",-1,1);
   \PI2 = Plot("g",-1,1);
 \end{plot2d}
\end{centerline}
\DoString("upper semicircle in red","UpperCircleRed");
\DoString("upper semicircle in white","UpperCircleWhite");
\DoString("background color green","BgGreen");
\DoString("background color blue","BgBlue");
\end{document}
```


Below this text a circle is shown, whose properties can be changed by using the buttons at the bottom.
upper semicircle in red
background color green
upper semicircle in white
background color blue

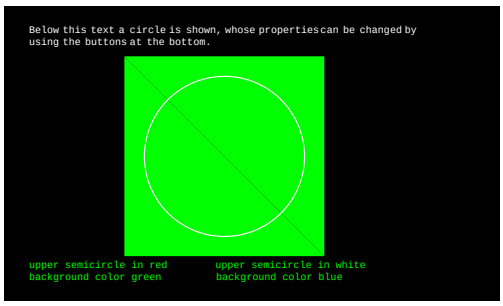
(a) Start


Below this text a circle is shown, whose properties can be changed by using the buttons at the bottom.
upper semicircle in red
background color green
upper semicircle in white
background color blue

(b) after clicking ”upper semicircle in red”


Below this text a circle is shown, whose properties can be changed by using the buttons at the bottom.
upper semicircle in red
background color green
upper semicircle in white
background color blue

(c) after clicking ”background color green”


Below this text a circle is shown, whose properties can be changed by using the buttons at the bottom.
upper semicircle in red
background color green
upper semicircle in white
background color blue

(d) after clicking ”upper semicircle in white”

Fig. 4.4: Redrawing plots depending on user interaction

4.1.4 Axes

Axes and its properties can be modified in a variety of different ways. First, `PlotAxes` defines which axis should be included in the plot.

Syntax: `\PlotAxes`

`\PlotAxes ="1"` adds the x-axis and `\PlotAxes ="2"` adds the y-axis to the plot. To draw both axes write

```
\PlotAxes = "12";
```

and for no axes at all

```
\PlotAxes = "";
```

As before, the color of the axes can be changed using the property `AxesColor`. Another important property is the range we want to display. Without further specifying it, the range is determined by the function or the vector we are going to show. If we only want to draw a certain part, or need a larger graphic to depict the function or vector, we can manually define the range using `AxisRange` and `AxesRanges`. The syntax is as follows:

Syntax:
```
\AxisRange(axsNr,min,max);
\AxesRanges(ranges);
```

The argument `axsNr` is Integer (1 or 2), the arguments `min` and `max` give the minimum and maximum of the corresponding axis interval. The second variant takes a 2x2-array, containing the bounds for the x- and y-axis. The command `GetAxesRanges` saves the minimal and maximal values on the corresponding axis in an 2x2 array, which is accessible by the specified variable identifier `A`.

Syntax: `\GetAxesRanges(A);`

To place automatic tick marks and labels to the axes we use `AutoTicks`. The syntax is analogous to `PlotAxes`.

Syntax: `\AutoTicks`

If we want to partition the axis manually, we can use the following commands. The placement of tick marks can be specified using the `PlotTick`.

Syntax:

```
\PlotTick(axsNr,position);
\PlotTick(axsNr,first,step,numTicks);
```

The first variant puts a tick mark at position `position` on the axis with number `axsNr` and writes the corresponding value as text label. The second variant puts tick marks and tick labels at the positions `first+k*step`, where `k` runs from 0 to `numTicks`-1. The color of the tick marks is determined by the axis color, the text label is given in the current text color. The precision of the numbers in the tick labels can be adjusted by setting the `Digits` property (see Section 3.2). Its integer value equals the number of displayed digits. Setting `Digits` to negative values, no text is displayed. Besides the tick label, we can adjust the way the tick mark looks. Using `TickLengthFactor` we can scale the default length of a tick mark. Using a negative value, the tick is placed on the other side of the coordinate axis.

Example 4.4 To understand the effect of the previously introduced commands, we want to have a look at the following source code. The result produced by SigMath is depicted in Figure 4.5.

Code: (PlotTick.txt)

```
\begin{document}
\begin{plot2d}
  \AutoTicks = "";
  \AxisRange(1,0,10);
  \AxisRange(2,-4,10);
  \PlotTick(2,-4,1,15);
  \PlotTick(1,2);
  \Digits = -1;
  \PlotTick(1,4);
  \Digits = 1;
  \TickLengthFactor = -4;
  \PlotTick(1,6);
  \TickLengthFactor = 2;
  \PlotTick(1,10);
\end{plot2d}
\end{document}
```

Finally, all objects like axes and labels have their own depth which can be modified by `AxesDepth` (see `TextDepth` in Section 2.5).

If we want to add additional objects or text to mark certain positions at the text we can also use the `AxisPos` command to detect the position of the axis on the screen.

Syntax: `\AxisPos(axisNr);`

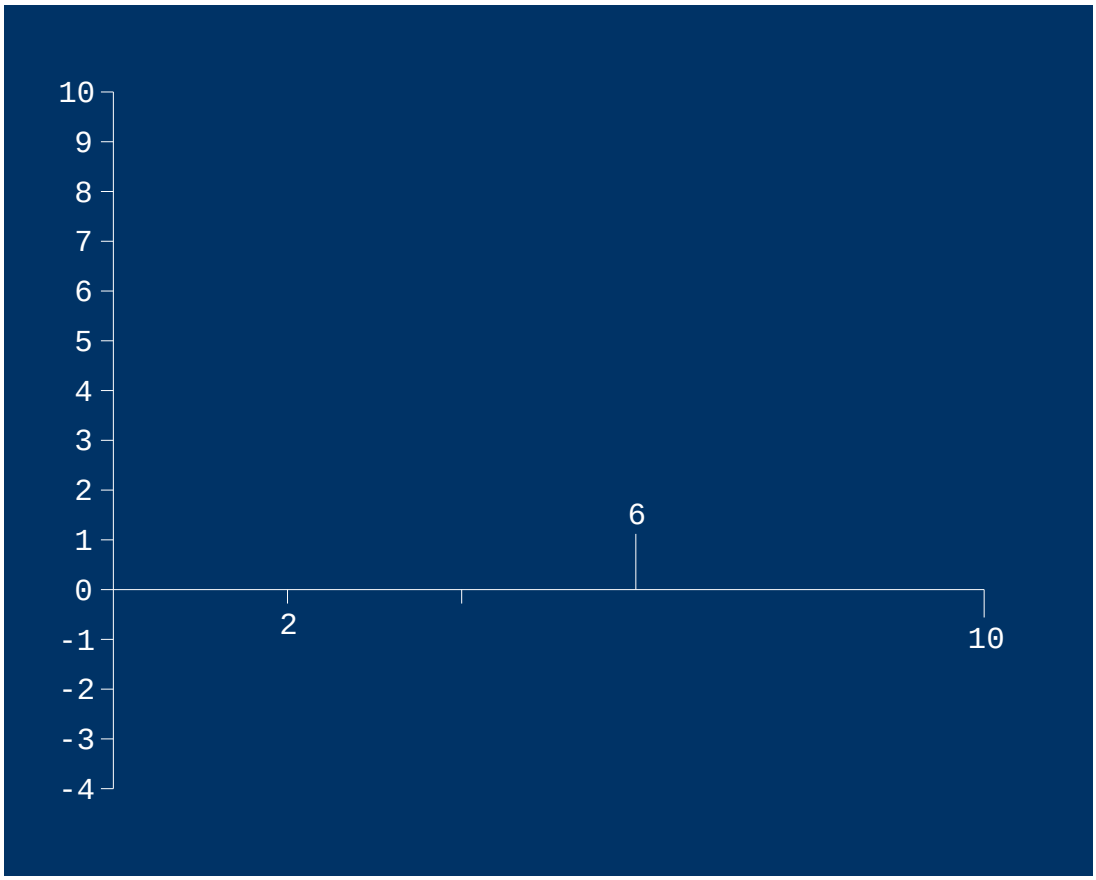


Fig. 4.5: Placing tick marks manually

It returns the second coordinate of the first axis if `axisNr = 1` and the first coordinate of the second axis if `axisNr = 2`. For other values or if no plot object is active, an error message is returned.

4.1.5 Interactive plots

SigMath's overall design principle is animation and interaction. Thus, plot objects can be modified interactively, too. To enable this, one has to set the property `Interactive` to `true`.

Syntax: `\Interactive = true;`

This allows the user to move the plot window on the screen by pressing the left mouse button and moving the mouse. Horizontal movement of the mouse amounts to horizontal scrolling of the visible x-axis area and leads to automatic computation of the corresponding vertical axis area. By vertical movements one can zoom in or out, yielding an interactive stretching or squeezing of the part of the x-axis. To limit the effect of interactive modifications, one can use `AxisMaxRange`.

Syntax: `\AxisMaxRange(axsNr,min,max);`

After issuing this command horizontal or vertical scrolling only changes the axis range of axis `axsNr` within the minimal value `min` and maximal value `max`.

Additionally, one can fix the displayed axis ranges by `AxisRange`.

4.1.6 Text

If we want to add text to a plot, we use the `postext` environment (see Section 2.5). The coordinates of the corresponding `TextPos` command are interpreted with respect to the plot coordinate system, i.e., with respect to the (not necessarily visible) coordinate axes. If we want the system to interpret the coordinates with respect to the plot window (relative to width and height of the plot) we set the property `TextPosPlotRect` to `true`. Then, the lower left corner has coordinates (0,0), the top right corner (1,1).

Another important topic is font selection for text and axis labels. Especially, if one wants to export a SigMath generated plot one has to take care. The position of a text block is computed using so-called font metrics, i.e., with respect to the chosen font family, font size, and font shape. SigMath is using its own font metrics which might differ from the ones used in, e.g., Adobe Reader. If one opens exported graphics in an external viewer application, the text is positioned at the location used by SigMath but maybe using different font metrics. This can result in shifts or overlapping. To prevent such effects, we suggest to use `monospace` fonts for such plots. More information can be found in the Chapter 5 on p. 67.

4.1.7 Summary

Table 4.1 contains all properties we introduced in this section in connection with two-dimensional plots. It serves as a short reminder. More information is contained in the online help of SigMath.

4.2 Three-dimensional plots

Property	Admissible values	Default value
`PlotWidth`	between 0 and 1	1.0
`PlotHeight`	between 0 and 1	1.0
`PlotLineWidth`	positive integer	1
`PlotLineStipple`	factor and `String` of length 16 consisting of ”0” and ”1”	undefined
`PlotPointSize`	positive integer	1
`PlotMargins`	array of 4 entries, each entry between 0 and 1	(0.1,0.1,0.1,0.1)
`TickLengthFactor`	arbitrary number	1.0
`PlotNumPointsX2D`	positive integer	128
`PlotNumPointsX3D`	between 0 and 1	32
`PlotNumPointsY3D`	between 0 and 1	32
`Interactive`	`true` or `false`	`false`
`PlotAxes`	`String` consisting of at most 3 digits	“123”
`AutoTicks`	`String` consisting of at most 3 digits	“123”
`PlotStyle`	`LINE_STRIP`, `FILL`, `RECT` or `FILLED_RECT`	`LINE_STRIP`
`PlotBgStyle`	`NONE`, `RECT` or `FILL`	`NONE`
`PlotColor`	color in RGB representation	white: (1,1,1)
`PlotBgColor`	color in RGB representation	black: (0,0,0)
`AxesColor`	color in RGB representation	white: (1,1,1)
`AxesDepth`	between -1000 and 1000	0.0
`PlotDepth`	between -1000 and 1000	0.0

Tbl. 4.1: Plot properties

5 Exporting graphics

This chapter deals with exporting graphics generated in SigMath to other file formats. The necessary functions are based on the GL2PS package by Christophe Geuzaine. To export figures little programming knowledge is necessary since SigMath provides a graphical user interface (GUI) for that task. SigMath allows to export to various typical file formats such as, e.g., PDF, PostScript. One can hide or uncover certain parts of the figure. If the figure is generated from an animation or can be modified by the user the figure can be "polished" before exporting it. This way SigMath allows to generate high quality figures to be used in other applications. Problems might occur in the precise positioning of text. Text might slip or be squeezed due to different letter sizes. In such cases one has to adjust the positioning in SigMath based on the later file format.

In order to add the export GUI to a document one adds the following two lines to the preamble of the document:

```
\variable ZS_FormatingDir = ProgDir + "\Shared\GraficsExport\Formating";
\Include(ZS_FormatingDir + "\Preamble.txt");
```

Here, `ZS_FormatingDir` contains the path to the directory where all necessary applications for graphics export are located. The second line includes the file `Preamble.txt` which contains further functions and variables that are working in the background. Next, add the following line to the page of the document that should be exported:

```
\Include(ZS_FormatingDir + "\ParameterPages.txt");
```

As soon as the user reaches this page the SigMath graphics export GUI appears. We provide a screenshot of this GUI in Figure 5.1. If the document consists of one page, only, put the above line directly following `\begin{document}`. If we have a multipage document structured by `\newpage; \label{labelOfPage};` we insert the line directly after the `newpage` command. All menu items are listed in Table 5.1.

In the following pages we introduce the available options. We can specify the target format, font families of text in figures, sort- and coding methods, and further general options. Some file formats allow to save additional title and producer of the file.

Format The most important choice you have to make in exporting graphics, is the format you want to save it to. This choice affects which applications can open the saved file, and the

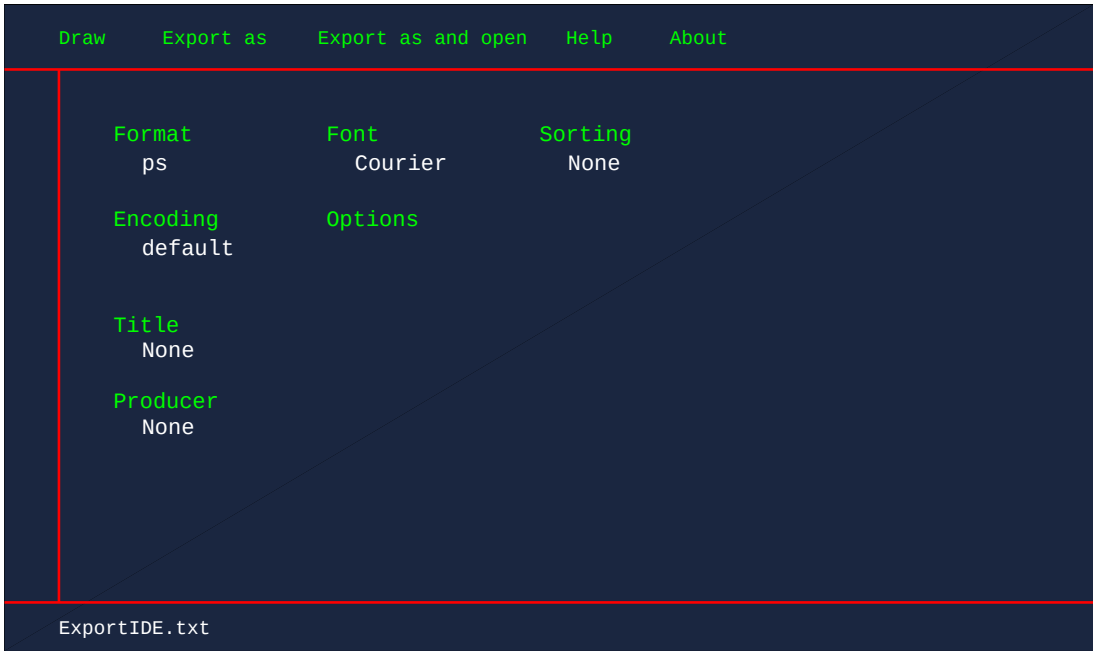


Fig. 5.1: The graphics export GUI

Menu item	Function
Draw	generates the figure without export
Export as	opens the ”Save as” dialog window; then enter the file name
Export as and open	saves the figure as in ”Export as” and opens the default application for the corresponding file format
Help	provides help and further information on the options
About	provides additional information

Tbl. 5.1: Menu items of the graphics export dialog window

possibilities of post-processing the data. All available graphic formats are given in Table 5.2.

If one does plan to insert the figure into further documents PostScript or PDF are the formats of choice. For usage in webpages one should preferize SVG files since most web browsers can display those embedded graphics directly (or via plugins).

Graphic import in LaTeX documents needs a little bit more explanation. The easiest way is to export to EPS format and directly include it into the LaTeX document. To do this, one has to load the corresponding packages in the preamble of the LaTeX file:

```
\usepackage[dvips]{graphicx}
```

Next, one includes the figure by

File format	Description
ps	PostScript
eps	Encapsulated PostScript (PostScript with further features needed for, e.g., LaTeX
pdf	Portable Document File
svg	scalable vector graphics in XML syntax; suitable for usage in web pages.
pgf	Portable LaTeX Graphics Format
tex	generates a LaTeX-file

Tbl. 5.2: Supported file formats in the graphics export

```
\begin{figure}[parameter for positioning]
\includegraphics[parameter on figure size]{Filename.eps}
\caption{caption} \label{label for later reference}
\end{figure}
```

This is a fast way to export SigMath graphics and include them to LaTeX documents. Another option is to leave text formatting to LaTeX. To do this, change the option "No text" in "Options" to "On". The code remains unchanged. Next, we generate an eps-file by specifying the corresponding output format. Now as usual, `\includegraphics` adds the figure to the LaTeX document. Another alternative is to generate the corresponding LaTeX code, directly. This contains the text of the figure and the import of the corresponding eps file. Be careful on hiding text: inserting axis labels at the right position might be a tedious thing to do. Thus, spending some time on the decision which application to use to insert text may save some work. Most of the time, exporting the whole figure to eps and include it to LaTeX in the described way will generate acceptable results.

Font This option allows to select the font family for the text in the figure. This can also be done in the source code by issuing the `gl2ps.FontName` command. If positioning of text plays a role, we recommend using the `monospace` font Courier. This increases the chance of matching font metrics which are used to calculate the positions of text blocks.

Sorting Using this option one can select an algorithm to detect the position of an object with respect to the observer, and hence, which objects cover other objects. Suitable for most 2D-graphics is the option `NONE`, for most 3D figures `Simple` will produce acceptable results. The most accurate, but computationally most extensive algorithm can be selected by `BSP`. We suggest to start with the low level settings and increase complexity until the resulting figure matches the requirements.

Font family	Sample text
Courier	Courier
Courier-Bold	**Courier bold**
Courier-Oblique	*Courier oblique*
Courier-BoldOblique	***Courier bold oblique***
Times-Roman	Times Roman
Times-Bold	**Times Roman bold**
Times-Italic	*Times Roman italic*
Times-BoldItalic	***Times Roman bold italic***
Helvetica	Helvetica
Helvetica-Bold	**Helvetica bold**
Helvetica-Oblique	*Helvetica oblique*
Helvetica-BoldOblique	***Helvetica bold oblique***
Symbol	symbols
ZapfDingbats	✓, ♥, ✕, ✠

Tbl. 5.3: Font families for the graphic export

Encoding This option only takes effect for EPS or PS. There are two states, `default` and `german`. Using `german` the character table is enlarged to display German umlauts and "ß" correctly.

Options Here, some further options can be enabled or disabled. A list is contained in Table 5.4.

Title For PostScript files one can add a title.

Producer For PostScript files one can specify the producer to be stored in the target file.

After treating all options of the export dialog we want to illustrate some subtilities. It might be necessary to hide interactive elements that are needed to adjust the figure and that should not be visible in the exported figure any more. This is realized by using an virtual `env` environments, which is not executed on export.

Code:

```
\begin{env}
\EnvBreakIf(gl.Get(gl.RENDER_MODE) != gl.RENDER);
    text and figure that are not to be exported
\end{env}
```

This code is also available as a template.

Option	**Description in case of state `on`**
occlusion cull	all hidden polygons are removed from the target file (reduces file size)
landscape	the target file is in landscape (instead of portrait)
no text	all text is hidden, see the remarks concerning LaTeX export.
draw background	the background is added
tight bounding box	the bounding box is put as close as possible around all exported objects
simple line offset	to all lines a small offset is added in z-coordinate (anti-aliasing)
no pixmap	all pixmaps are removed from the target file
best root	the construction of the `BSP`-tree is optimized by using the root-primitive yielding minimal number of branches
no ps3 shading	only on PostScript; disables the usage of shfill (PostScript level 3 operator) that enhances the visualization of smooth shaded primitives but might complicate later PDF conversion
no blending	opaqueness is disabled disregarding any commands in source code
smooth shading	only on PostScript or SVG; controls the number of flat shaded triangles that are used to approximate smooth shaded triangles; needs three arguments `(nr,ng,nb)` which specify how many values (with respect to the red, green, and blue color components) are used in the interpolation; a triangle is subdivided recursively until the color difference (in the color components) on its corners is less than `1/nr`, `1/ng`, and `1/nb`, respectively

Tbl. 5.4: Advanced options in graphics export

After exporting a SigMath document `test.txt` two files are generated. Besides the exported graphics file (e.g., `test.eps`) a file `test.options` containing all options used to generate `test.eps` can be found in the corresponding directory. If one opens `test.txt` in SigMath again, all options are automatically loaded from `test.options`.

To save one frame in a running animation, we stop the animation by pressing *CTRL + Space* and open the Export dialog for the static figure by *CTRL + S*.

6 Programming

This chapter treats the programming language of SigMath. We introduce all fundamental structures in SigMath. Section 6.2 deals with properties and methods of all implemented data types, Section 6.4 contains information on control structures, and Section 6.5 deals with functions. We close this chapter with a section on debugging tools in SigMath.

6.1 Classes

Commands in SigMath are grouped within certain classes. To tell SigMath's interpreter where it can find the definition of the used function, we have to provide the name of the class, the command is contained. Then the interpreter looks up the definition of the command and checks whether all formal requirements for execution of this command are met, i.e., if all required arguments are specified or all arguments are of correct type. We give a list of all classes in SigMath in Table 6.1. The most important classes and commands are then described in more detail in the following sections.

By default, the class `doc` is activated, i.e., we do not have to specify this class explicitely. To use functions contained in other classes, we have to use *dot notation*, i.e., we have to provide the name of the class in front of the command, separated by a dot. Thus, the general syntax for a command looks as follows:

Syntax: *`class.command`*`;`

In particular, whenever we are using complex data types (see Section 6.3) we have to use this notation. If a document contains a lot of commands from one class, we can preselect the corresponding class by writing `use_class className`. Then the class `className` is activated, i.e., the interpreter is parsing `className` instead of `doc`.

Syntax: `\use_class NameOfClass;`

For example, in parts of a document containing a lot of commands which are given by the OpenGL class we can put `use_class gl;` in front and do not have to specify `gl.` every time we use a command from that class. This also works for dynamic classes, which are imported from DLL-files (see Chapter 8).

Apart from the classes, SigMath also contains some globally defined functions and identifiers. Those are, e.g., the names of data types, mathematical functions and commands to generate

Class	Description
anim	Animation
array	Arrays
bmp	Bitmaps
cont	Container
doc	Text formatting
edit	Text editor
font	Font
ge	Graphics export
gl	OpenGL
glpoint	3D- and 4D-Points
glu	OpenGL utility library (GLU)
gluti	OpenGL help
ui	Graphical user interface
re	Renderer
string	String
syst	System
thread	Threads

Tbl. 6.1: Classes in SigMath

random numbers.

Finally, we distinguish between two types of commands that are related to certain class properties. If the command can only be used to request the current value of the property it is *read-only*. Accordingly, a command is *write-only* if we can only assign a new value to a property, and the command has the *read-write* property if both actions are possible.

A comprehensive list of all methods and properties of the various classes in SigMath is given in the online help.

The anim class

The class `anim` contains commands connected with animations. Since they are fundamental but easy to understand, we introduce them here.

For screen design, there are five properties available, `Height`, `Width`, `Ratio`, and `ScreenHeight` and `ScreenWidth`.

Syntax:

```
\anim.Height
\anim.Width
```

```
\anim.Ratio
\anim.ScreenWidth
\anim.ScreenHeight
```

Those properties give the height and the width of the SigMath window in pixels. The third property gives the ratio of the first two properties. The last two properties give the height and width of the screen in pixels.

Apart from that, two further properties are connected with SigMath's animation history. First, `FrameNr` counts the number of frames SigMath computed and displayed since the document has been started.

Syntax: `\anim.FrameNr`

The property `Time` measures the time an animation is running in seconds.

Syntax: `\anim.Time`

If the user pauses an animation by pressing *CTRL + Space*, `Time` and `FrameNr` remain constant till the animation is resumed. This can be used to restart animations after such a break at the correct time position. The refreshment rate of the screen can be adjusted by the property `FramesPerSec`. Mind that, the actual refreshment rate depends on the number and type of calculations necessary, the overall system, and the current system load.

Syntax: `\anim.FramesPerSec`

Finally, the property `Busywaiting` controls SigMath's waiting behavior.

Syntax: `\anim.Busywaiting`

After finishing all steps to display the next frame, SigMath either runs an (nearly empty) loop (`Busywaiting=true`) or uses the *Sleep* function of the operating system until the frame is displayed.

6.2 Data types

SigMath distinguishes between two kinds of objects, constants and variables. Constants (see Section 6.2.1) have a fixed value and no identifier. They are sometimes also called literals. Variables (Section 6.2.2) act as a placeholder for such values. Such an object can change its "content" in course of time. Every such object has a specified type, the *data type* (Section 6.2.3) and, thus, inherits all properties of this class. Every data type has a number of properties and functions to modify them. For example, numbers can be added, text can be concatenated or we

can compare boolean values. Finally, more complex objects can be built from elementary data types, e.g., matrices of numbers which allow to simplify certain tasks (Section 6.3).

6.2.1 Constants

If we speak about a constant, we typically mean its concrete value, e.g., the number 5 or the text "Hello world!". Constants have to be treated as their actual values. They can not be changed. Table 6.2 shows some examples for constants.

Data Type	Example	Description
`Integer`	`98`	Decimal
	`0x12, 0X12`	Hexadecimal (16)
	`012`	Octal (8)
`Real`	`1.2, -2.6`	
	`11e+2, 11E+2`	$11 * 10^2 = 1100.0$
	`Pi`	$3.14159\ldots$
	`e`	$2.71828\ldots$
	`INF, -INF`	plus or minus infinity
	`NAN`	Not-a-Number
`Complex`	`I`	imaginary unit
	`5.0 + 2.5*I`	
`Bool`	`true, false`	true, false
`String`	`"some text"`	

Tbl. 6.2: Constants

6.2.2 Variables

Variables are used to store data temporarily in memory. For this purpose, different storage templates exist which are called data types. Those storage templates allocate memory and assign names to the memory areas for future read or write access. Internally, memory is organized in (memory) addresses, which are then linked to the name of a variable. An overview of all data types used in SigMath is given in Table 6.3.

Initialization of variables

To make a variable accessible within a SigMath documents, it has to be defined. SigMath (not only allows, but) demands simultaneous initialization of the variable, i.e., we (have to) assign an initial value to it.

Syntax: `\variable NameOfVariable = InitialValue;`

Identifier	Description	See pp.
`Integer`	integers	78
`Real`	floating point numbers	78
`Complex`	complex floating point numbers	78
`Bool`	boolean values	78
`Void`	empty data type	79
`Pointer`	memory addresses	79
`String`	string of characters	79 and 84
`Arrays`	arrays	92
`Container`	complex data structures	102
`GLPoint`	4-dimensional vector	99
`Bitmap`	color values of an image	105
`TextEditor`	input field with the functionality of a text editor	105

Tbl. 6.3: Data types in SigMath

The reserved word `variable` tells SigMath to interpret the following as a definition of a variable. `NameOfVariable` is a name which can be chosen (according to some rules) by the programmer. The `InitialValue` can take different shapes. The simplest variant is to specify a constant. Basically, all SigMath expressions that return a value of fixed data type (see Section 6.2.4) are valid. Initialization defines the data type of the variable. Therefore, SigMath allows great flexibility, which of course, has to be handled with care. For example, the initial value can be specified using a function, whose returned data type is specified on execution, only. Let us consider some examples for initializations and their consequences for the data types.

```
\variable r = 5.0;                              // r is variable of type Real
\variable x = 5;                              // x is variable of type Integer
\variable y = 2 + x;            // y is initialized by x+2 (here 7) and thus
                                   // recognized as variable of type Integer
\variable t = "Hello World!";               // t is a variable of type String
```

The value of the variable `y` obviously depends on the value of `x`. This expression is evaluated only once and assigned to the variable `y` as initial value. Thus, changing the value of `x` afterwards does not lead to a change of the value of `y`. The reason for that is simple: after definition and initialization the variable `y` has the value `7`, which does not depend on `x` any more. The variable is thus assigned the *resulting value* of the expression and *not* the expression itself.

If a variable is used as an argument of a function there is one exception from the automatic type determination. Sometimes, we have to declare the data type, explicitely, see Section 6.5.

The identifiers of variables have to be chosen according to the following rules:

1. Names may contain all Latin characters, the digits `0` to `9`, and the underscore `_`.
2. The first character has to be a letter.
3. Identifiers must not be identical to a reserved word of SigMath (e.g., `variable`). Table 6.4 contains a list of all reserved words in SigMath.

Reserved Word	Description
`case`	part of the `switch-case` structure
`default`	optional part of the `switch-case` structure
`e`	Euler number `e = 2.71828...`
`else`	part of the `if-else` structure
`false`	Boolean constant
`for`	part of the `for` loop
`function`	definition of a function
`I`	imaginary unit
`if`	part of the `if-else` structure
`INF`	constant infinity
`NAN`	constant for Not-A-Number
`Pi`	`Pi = 3.14159`
`switch`	part of the `switch-case` structure
`true`	Boolean constant
`use_class`	specifies a certain class
`variable`	definition of a variable
`while`	part of the `while` loop

Tbl. 6.4: Reserved words in SigMath

The name of a variable can have arbitrary many characters and is case-sensitive. Although there are basically no limitations to our creativity, we suggest to use ”speaking” names and care about uniqueness. One should not define two variables having the same name. Only in very limited cases, as, e.g., for the counting variable in a loop (see Section 6.4.2) one can use the same name for different (locally valid) variables in different environments.

Apart from uniqueness and meaning one should be consistent. Names of variables or functions in this guide always start with an uppercase letter. Several words in one name are easily separated by using upper case letters (`MoreThanOneWord`). Only names consisting of one character only are excluded (like, e.g., `x`). Arguments in function heads begin with lower case letters (`formalArgument`). Thus, the way a name is spelled already contains information on the meaning and usage of the variable.

To determine the data type of a given variable, we use the `TypeOf` function. It returns the data type of its argument.

Syntax: `\TypeOf(argument)`

As we will see later, one does not have to specify the data type of an argument of a function in the definition. Thus, within the function body, one can use `TypeOf` to determine the data type (see Section 6.5).

Accessing and displaying variables

Since nearly all entries in a SigMath document are interpreted as text, we have to use special commands to tell the interpreter to, e.g., write out the value of a variable and not just its name or to assign a new value to a defined variable. To access a variable `x` we simply put a backslash `\` in front of the variable outside of `code` environments. To display the value of a variable, its internal format has to be converted to a displayable form, e.g., numbers have to be converted from their internal number format to a sequence of digits or boolean values have to be rephrased as text. All this conversions are performed by `ValueOf`.

Syntax: `\ValueOf(varName);`

The argument `varName` is of type String that contains the name of the variable to be displayed. As an example, if we define the variable `x` and initialize it to `0`, we display its value by writing the following line of code:

```
\variable x = 0;
x = \ValueOf("x");
```

By that, a line containing the text `x = 0` is displayed on the screen, since `x` and the equation sign are interpreted as simple text.

Scope and visibility of variables

A special property of any variable, besides its data type, is its scope and visibility area. The scope of a variable is given by all environments which allow read and write access to the variable. The visibility area is concerned with names of variables that are used several times. Since scope and visibility also apply to functions all statements in this section also apply to functions.

With regard to scope of a variable there are two options: either the scope of a variable is the whole document or an environment. In the first case the variable is called global, in the second case its local. Variables that are defined within an environment are no longer accessible as soon as the defining environment is left. All functions and environments we encountered in section 2.1 limit the scope of a variable!

Local variables of an environment are valid in all sub-environments. The only exception to this rule connected with nested environments is the following: every time a ”global” and a local

variable of the same name are specified, the local variable *hides* the global one. The global variable remains *valid*, but is no longer *visible*, since it is hidden by the local one. This guarantees uniqueness of variable names at each time point of code execution since this rule prohibits the visibility of two variables with the same name.

In order to prevent the described effect we suggest to define variables "as local as" possible. Thus variables of the same name may exist during execution of the code but not simultaneously.

6.2.3 Elementary data types

Every data types comes along with specified memory templates. This template organizes how much memory (bytes) is allocated and how to interpret it. The number of bytes determines the range of the variables, interpretation of 0 and 1 gives a meaning. Thus, a 1-Bit is interpreted as `true`, if the associated variable is of type `Bool`, in case of an `Integer` variable the interpretation would be the number 1. Besides there are data type specific operators. Elementary data types are the building blocks for higher, more complex data types (see Section 6.3).

Arithmetical data types

The arithmetical data types are given by `Integer`, `Real`, and `Complex`. The range of an Integer is a subrange of all integers, whereas data of type `Real` are floating point numbers. The set of all floating point numbers is a finite subset of the set of all rational numbers. The data type `Complex` represents the complex numbers and contains the constant `I` for the imaginary unit. Any value of type `Complex` is stored as a pair of two floating point numbers. The notation for constants is given by `c = a + b*I`, where `a` and `b` are of arithmetical data type. By using the complex unit `I` SigMath interprets this as a complex number. Mind that even if a complex number `c` is specified in the form `c = 5 + 2*I`, it is treated as a pair of floating point numbers, and not as a pair of `Integer` values. Table 6.5 gives the range of the data types and the required memory sizes.

Data type	Range	Memory size
`Integer`	-2147483648 to 2147483647	4 bytes
`Real`	$\pm 1.79769 \cdot 10^{\pm 308}$	8 bytes
`Complex`	$\pm 1.79769 \cdot 10^{\pm 308} \pm 1.79769 \cdot 10^{\pm 308} i$	16 bytes

Tbl. 6.5: Ranges for variables of type `Integer`, `Real`, and `Complex`

Boolean variables

Boolean values have the identifier `Bool` in SigMath. They can only take the following two values: `true` or `false`. The data type `Bool` often appears as the result of an expression of the form `5 > 10`. Typically, we use boolean values in connection with `if` conditions, see Section 6.4.1.

Void - the empty data type

Variables of type `Void` can not be defined explicitly in SigMath. This data type is only used internally, to "simulate" a return value of functions that are not returning any value. If one writes a function, that should change the value of a global variable, we do not need a return value. Such functions implicitly define a return value of type `Void`.

Pointer

The data type `Pointer` is used to store (memory) addresses. It is only used in connection with DLLs, see chapter 8. To define a variable of type `Pointer` we write `Pointer()`.

Syntax:

```
\variable P = Pointer();
\variable Q = P;
```

Both commands generate a `Pointer` which points to the zero-address.

The comparison operators `==` and `!=` are also available for `Pointer`. Also comparison of `Pointer` and `Integer` variables are possible. One can use

```
\P == 0
```

to test if `P` directs to the zero address.

Strings

Since variables of type `String` are fundamental objects in SigMath's programming language we introduce them in this section on elementary data types. Nevertheless, it is in fact a complex data type. We will come back to its interpretation as a class in Section 6.3.1.

In contrast to arithmetical data types a value of type `String` is used to present sequences of characters of arbitrary length. Those sequences of characters are delimited by quotation marks `""`. Since those delimiters may enclose character sequences of arbitrary length the data type `String` allows easy manipulation of text. Let us give some examples for the initialization of variables of type `String`:

```
\variable S = "";
\variable T = "Hello!";
\variable V = "A short text with blanks!";
```

Since the length of a `String` variable can change, one cannot allocate a fixed amount of memory beforehand. Its memory consumption is dynamically adjusted to the value of the variable, i.e., changing the value during run time changes the amount of reserved memory.

6.2.4 Operators and expressions

To handle variables according to their data types we use operators or functions (see Section 6.5). Operators are predefined functions which allow for a different syntax. The arguments are (typically) not given after the name of the operator in parenthesis, the operator is written as a short symbol, and in case of binary operators, the operator is placed in between the two arguments. One simple example for an operator for `Integers` is the addition operator "`+`". Arguments of an operator are typically called *operands*.

An *expression* is a concatenation of variables, constants, and function calls, which evaluates to a certain value. Every such part is an expression on its own and combinations of expressions are expressions. All expressions determining a data type on evaluation can be used to initialize variables (see Section 6.2.2). In particular, that includes function calls or expressions connected with operators can be used for that.

Before introducing the available operators in SigMath, we fix some terminology. Depending on the number of operands, we distinguish *unary* and *binary* operators. Unary operators have one operand, only (e.g., sign operator) while binary operators, such as the addition, need two operands. Every operator has a *priority*, which allows to define an order of execution within one expression. The highest priority is 1. As is known from elementary calculus, the multiplication operator has a higher priority than the addition operator.

Basically, there are two ways to manipulate the order of execution determined by the priorities of the operators. The first way is to enclose subexpressions by parenthesis. Then the operands in the expression in parenthesis are executed first, where their execution order within the subexpression is determined by the priority of the operators. A second options is to enclose parts of the expressions in functions.

In the following we study the effect and some specialties of the operators listed in Table 6.6.

Sign

The sign operator is an unary operator and should not be confused with the addition or subtraction. The operators `+` and `-` add a sign to variables of arithmetical data type. A positive sign does not have to be written explicitely.

Arithmetical operators

The arithmetical operators `+`, `-`, `*`, `/`, and `^` are all binary operators. Their operands may be any variables of arithmetical data type.

If all operands belong to the same data type the effect of the arithmetical operators is intuitively clear. Addition, subtractions, multiplication, division, and exponentiation are executed within the data type. Division of two integers produces the integer part of the result, if the result would be rational. Thus, e.g., `5/2=2` and `5.0/2=2.5`.

Priority	Operator	Description	Example
1	`++ --`	increment, decrement	`++A`
1	`+ -`	sign (unary)	`+5`
1	`!`	bitwise complement	`!5`
1	`!`	logical negation	`!B`
2	`^`	exponentiation	`5.0 ^ 2.0`
3	`* /`	multiplication, division	`5 * 2`
4	`+ -`	addition, subtraction	`5 + 2`
4	`+`	concatenation	`"abc" + str`
5	`< <=` `> >=`	smaller, smaller or equal greater, greater or equal	`i < 5` `i >5`
6	`== !=`	equality, inequality	`i == 5`
7	`&&`	bitwise AND	`5 && 2`
8	`||`	bitwise OR	`5 || 2`
9	`&&`	logical AND	`B1 && B2`
10	`||`	logical OR	`B1 || B2`
11	`=`	assignment	`i = 5`
11	`*= /=` `+= -=`	compound assignments	`x *= 2` `x += 2`

Tbl. 6.6: Operators in SigMath

For mixed expressions, i.e., where the two operands belong to different data types, the basic behavior is as follows: the expression returns a variable of the more ”complex” of the data types of the two operands. Thus, automatic type conversion is performed resulting in the listed return data types in Table 6.7.

`operator`	**Integer**	**Real**	**Complex**
Integer	Integer	Real	Complex
Real	Real	Real	Complex
Complex	Complex	Complex	Complex

Tbl. 6.7: Automatic type conversion for arithmetic operators

Another specialty is concerned with assignments. If the value of an expression is assigned to an (already existing) variable, two data type conversions may occur: a first conversion towards a more ”complex” data type may be needed due to the arithmetical operator; a second conversion might be necessary to store the result in the data type of the variable (which might be ”less complex” than the resulting type). Thus, for example if an arithmetical operator yields a result of type `Real` and the target variable is of type `Integer`, the decimal places of the `Real` variable are cut off and only the resulting integer part is assigned to the `Integer` variable.

Assignments

In a simple assignment, the operator `"="` sets the value of the left operand, which has to be a variable, to the value of the right operand. If both operands are of the same data type the assignment is directly executed. Otherwise, SigMath tries to convert the value of the right operand to the data type of the second variant. Type conversions of the left operand is only possible during initialization of a variable. Thus, initialization determines the data type of the variable and acts as an assignment.

Besides the simple assignment operator Table 6.6 contains the compound assignments `+=`, `-=`, `*=`, and `/=`. This operators combine assignments with arithmetical operators. The new value of the variable on the left is given by its old value combined with the (arithmetical) operator and the value of the expression on the right hand side. For example, after `\x *= 2` the value of the variable `x` is the twofold of its former value. This allows a compact notation of source code and improves readability of the code.

Assignments operators are also available for complex data types, such as arrays. We return to that in Section 6.3.2.

Incremental operators

The increment operator "`++`" raises the value of the variable by `1`. It is an unary operator and is written in front of the name of the variable. Similar, the decrement operator "`--`" lowers the value of the variable by `1`. Mind that the variable has to be of type `Integer` or `Real`. Variables of type `Complex` are not allowed. Those operators are especially useful for counting variables and its usage may be preferred to `\i = i + 1` or `\i += 1`. The last variant has the same effect, though.

Comparisons

Comparison operators allow to compare the values of two expressions. The result of such an comparison is of type `Bool`. All available comparison operators are listed in Table 6.6 together with an example. Comparisons are used in conditions and loops and may be used to decide between two (or more) alternatives.

If one tries to compare mixed expressions, both expressions are automatically converted to the more complex data type then compared within this class in a mathematically sound way. The values `2` and `2.0`, for example, have different realizations and are not equal with respect to the system. After conversion SigMath recognizes them as the same number. On arrays different rules apply, see Section 6.3.2 for more information.

Logical operators

Logical operators allow to combine different Boolean values. The operands of a logical operator thus have to be of type `Bool`. Thus, also expressions which return a value of type `Bool` are

allowed.

There are three logical operators: The two binary operators AND (`&&`) and OR (`||`) combine two Boolean values, the unary operator NOT (`!`) negates the value. Mind that OR is not exclusive, i.e., as soon as one of the two operands has the value `true` the value of the `||` is `true`. We state all possible connections of Boolean values in Table 6.8.

&&	true	false
true	true	false
false	false	false

||	true	false
true	true	true
false	true	false

A	!A
true	false
false	true

Tbl. 6.8: Truth tables

Bitwise operators

Bitwise operators allow the direct manipulation of the coding of `Integer` variables. Using AND, OR and NOT operators, data elements of type `Integer` may be composed or negated. We use the same symbols as for the logical operators (`&&`, `||`, and `!`). A bit with entry `1` is then interpreted as `true` and with entry `0` as `false`.

Integers in SigMath always use 32 digits, thus every bit operator results in 32 comparisons or negations. The operator is performed bitwise.

`i`	=	8_{10}	= $0\ldots01000_2$
`j`	=	5_{10}	= $0\ldots00101_2$
`i && j`	=	0_{10}	= $0\ldots00000_2$
`i || j`	=	13_{10}	= $0\ldots01101_2$
`!i`	=	$-\ 9_{10}$	= $1\ldots10111_2$
`!j`	=	$-\ 6_{10}$	= $1\ldots11010_2$

Besides those there are two more bit operators which are given by the functions `BinaryShift` and `BinaryRotation`. They can be used to shift the bit pattern of an `Integer` variable to the left or to the right and thus change the value of the variable. Let us have a look at the syntax and a small example.

Syntax:

```
\BinaryShift(i, num);
\BinaryRotation(i, num);
```

`i` is the `Integer` variable we want to modify. The argument `num` gives the size of the shift. If `num`>0 we shift to the right, otherwise to the left. The two functions differ in the treatment of the beginning and end of the `Integer i`: shifting to the right, positions at the beginning are vacant and the corresponding number of bits on the right are cut. Shifting only fills up the

vacant positions by 0 and drops the bits at the end. Rotation copies the trailing bits from the end to the vacant part at the beginning.

Code	Binary representation	Decimal representation
`i`	00000000000000000000000000001001	9
`BinaryShift(i,2)`	00000000000000000000000000100100	2
`BinaryRotation(i,2)`	01000000000000000000000000100100	$2^{31}+2$

The framed parts of `i` can be found at the specified positions. While 01 is lost on the shift, it is placed at the beginning of the variable on rotation. 00 shows the filling with zeros. The effect of the commands is not *in-situ*, i.e., the value of `i` remains the same, the return value is placed in a different part of the memory and can be stored to a new variable by writing `j = \BinaryShift(i,2)`. There is no distinction between sign, mantissa or exponent. Thus, the sign of an Integer may be changed by bit manipulation. No type conversion is performed, i.e., all operands have to be of type `Integer`.

6.3 Complex data types

A complex data type is built from elementary data types and is internally realized as class in SigMath (see Section 6.1). In this case, classes act as templates. They contain properties and *methods*, i.e., functions that allow to manipulate the properties. A *variable of complex data type* is an instance of the class that has all properties and methods the class provides. The elementary components of a variable of complex data type are called *data fields* and can be interpreted as variables of elementary data type. Methods, data fields or properties of an instance of the class are called using dot notation (see Section 6.1) and, thus, do not require the variable as an input argument. There even exist methods of the same name, their "uniqueness" is guaranteed by class membership of the calling object in front of the dot.

SigMath provides the following complex data types: `array`, `cont`, `glpoint`, `string`, `bitmap`, and `edit`. Arrays provide the possibility to store homogeneous data in a vector, a matrix or some higher dimensional object. Containers are generalized list structures with additional sorting properties. The `GLPoint` data type stores coordinates in three dimensional space and allows to easily describe movements or changes of viewpoint by a fourth dimension. The `String` data type, already introduced in Section 6.2.3 is also a complex data type. It allows to modify sequences of characters far beyond the elementary concepts. We will provide more detail on all complex data types and describe their methods in the following sections.

6.3.1 Strings

The `String` data type is a storage template for sequences of characters, or shortly, text. Since text may have arbitrary length, we cannot allocate a fixed amount of space for such a variable beforehand and thus, sequences of characters require a complex data type. As most of the other programming languages, SigMath provides methods to manipulate `String` variables efficiently

and intuitively. We introduce the available methods and give reference to more general functions that are defined in the `string` class. Throughout, we assume a variable `S` to be initialized like in the following example:

```
\variable S = "";
```

The arithmetical operator `+` is also defined for operands of type `String`. It concatenates two objects of that type to a new object of type `String`. The compound assignment `+=` appends the right operand to the end of the left operand, while the right operand remains unchanged (compare Table 6.6 in Section 6.2.4).

```
\variable S1 = "Hello";
\S1 += " World!";
```

After execution of those two lines of code the variable `S1` has the value `"Hello World!"`. The reserved memory for a `String` variable has to be adjusted, dynamically: after initialization `S1` contains 5 characters and after concatenation and assignment its length has increased to 11 characters, occupying more memory than before.

We interpret objects of type `String` as sequences of characters. *Every* character, even a blank, has a uniquely determined position, its *index*, and can be accessed by the index operator `()`. `S(i)` returns the `i`-th character of `S`. Indexing starts with `1` and only positive integers are allowed as arguments.

The length of a `String` is given by the property `Size`. It is a read-write property, i.e., we can also assign a new value to this property.

Syntax: `\S.Size`

If the new length is shorter than the actual length of the `String`, the pending characters are cut off. If the length is larger, the text is filled up with blanks. If we assign a negative value, the length of the `String` is automatically set to 0.

Since we cannot simply add reserved symbols in SigMath to strings, the `string` class contains substitutes for such symbols. If we want to write a filename with its path to a variable, we would like to write, e.g., `S="Images\File.bmp"`. This would return an error message since \is a reserved symbol. The backslash has to be generated by `Backslash` and quotation marks are inserted by `Quote`, instead.

Syntax:

```
\string.Backslash;
\string.Quote;
```

Thus, the correct format of the above command is

```
\S = "Images"+ string.Backslash + "File.bmp";
```

Text formatting commands like line breaks or tabulators can be inserted by `n` and `t`.

Syntax:
```
\string.n;
\string.t;
```

Both commands increase the size of a string by `1`.

Sometimes it is necessary to detect, whether a given string `S` only contains numbers or letters. For this, the following three methods are available:

Syntax:
```
\S.IsAlNum;
\S.IsAlpha;
\S.IsDigit;
```

Here, `IsAlNum` return `true` if `S` only contains *alphanumeric* characters, i.e., numbers or letters. `IsAlpha` checks for letters, only and `IsDigit` checks for numbers.

Methods to modify strings

To extract a substring from a given String, we can use the method `SubString`.

Syntax: `\S.Substring(index[,count])`

Here, `SubString` returns the String starting at index `index` of `S`. The optional argument `count` specifies the length of the substring. If `index` exceeds the length of `S` or `count` is non-positive, then the empty String is returned.

The `SubstringBefore` method returns the substring in `S` that stands before the first appearance of the specified text.

Syntax: `\S.SubstringBefore(s)`

Here, `s` contains the ”separating” text, e.g., dot that separates the filename from its file name extension. The `SubstringBefore` method only works as a function and leaves `S` unchanged and the returning value has to be stored to another variable. If `S` does not contain `s` as a substring, the method returns the value of `S`.

The methods `Position` and `PositionCI` allow to search for the starting index of the first appearance of a substring.

Syntax:
```
\S.Position([index,]s)
\S.PositionCI([index,]s)
```

Here, `s` specifies the text to be searched for and the optional argument `index` allows to specify an index where the search starts in the `String`. As before, a negative index is automatically mapped to 0 and if `index` is larger than `S.Size` an error message is returned. The first variant `Position` is case-sensitive, the second `PositionCI` is case-insensitive.

To insert a new substring in an existing string, we can use `Insert`.

Syntax: `\S.Insert(index,s);`

Here, `index` has to be a suitable `Integer` value and `s` is the inserted text, i.e., an expression of type `String`.

The counterpart of `Insert` is `Delete`. This method allows to remove a number of subsequent characters from `S` starting from a specified index in `S`.

Syntax: `\S.Delete(index,count);`

Here, `index` has to be an `Integer` between 1 and `S.Size` which determines the beginning of the substring to be erased. The second argument `count` has to be a non-negative `Integer` which gives the length of the substring. If `count` is greater than the length of the remaining tail of `S` after position `index`, only this tail is removed without warning.

The `Replace` method can be used to replace substrings in a `String` by another substring.

Syntax:
```
\S.Replace(old,new);
\S.Replace(old1,new1,old2,new2);
```

Here, `old` is a `String` variable which is searched in `S` and is replaced by the value of the `String` variable `new`, every time it occurs in `S`. The second variant allows to replace every appearance of a substring of the form `old1 + s + old2` by `new1 +s + new2`, where `s` is any substring. Both methods return the number of replacements which have been performed. The second variant is particularly useful if one, e.g., has to exchange symbols for parenthesis efficiently.

The methods `ToLower` and `ToUpper` change all characters to lower or uppercase letters if possible. All other characters remain unchanged.

Syntax:
```
\S.ToLower;
```

```
\S.ToUpper;
\string.ToLower(s);
\string.ToUpper(s);
```

Here, the first two commands act on the `String S` directly. The last two functions leave `S` unchanged and return a modified `String` which has to be stored to another variable.

The `Trim` method removes all leading and closing blanks and all control symbols.

Syntax:
```
\S.Trim;
\string.Trim(s);
```

The first command directly acts on `S`, whereas the second variant works as a function and returns the modified string.

Methods to export strings

This paragraph contains methods to export and import text.

To import text from an existing text file, we can use the method `Load`.

Syntax: `\S.Load(filename);`

Here, `filename` is the full qualified path name of the file to be loaded (drive, directory, and filename; also see Table 1.3). The method returns `false` if the file is not found or cannot be loaded.

The counterpart of `Load` is `Save`. It allows to save the value of `S` to a text file.

Syntax: `\S.Save(filename);`

Here, `filename` is an expression of type `String`, which specifies the filename and target path of the file which afterwards contains the value of `S`.

If we want to load only parts of a file or save more than one string to a text file, we have to use the following two methods, instead.

Syntax:
```
\S.SaveToKey(filename,key[,leadingString]);
\S.LoadFromKey(filename,key[,leadingString]);
```

The `SaveToKey` method can be used to save the value of a string variable to a certain marked position (a key) in a file. Here `filename` is the name of a file (with full qualified path name). Without giving the optional argument `leadingString` the method writes the value of the variable `S` to the file. It is written in the file directly following the first line beginning with the

string #key#. With the optional argument leadingString SigMath searches in the file for the first line beginning with the expression leadingString + #key#. If the file does not contain a line beginning with #key# or leadingString + #key# SigMath appends a line with #key# or leadingString + #key# followed by the value of the String variable S. The method returns a Boolean value: true if filename exists and it contains a line with #key# and false, otherwise.

The method LoadFromKey searches in the file with full qualified path name filename for the first line either beginning with #key# or leadingString + #key#. Then SigMath writes everything following this line to S until it detects # or the file ends. If filename = "" SigMath ignores leadingString and searches the current document for an environment of the following form:

```
\begin{#key}
 some text
\end{#key}#
```

If the search is successful, SigMath writes the content of this environment to S. Additionally, LoadFromKey returns true if filename exists and it contains #key and false, otherwise.

If we want to exchange text data via the clipboard we can use the methods LoadFromClipboard and SaveToClipBoard.

Syntax:

```
\S.LoadFromClipboard;
\S.SaveToClipboard;
```

Here, LoadFromClipboard writes the whole content of the clipboard to the variable S, whereas SaveToClipBoard writes the value of S to the clipboard.

Methods for data type conversion

The following section introduces functions that are not acting on String variables, but are commands provided by the string class. An important group of commands is devoted to converting variables of different data types to String and the vice versa.

To convert variables to String variables we use the ToString function. It automatically detects the data type of its argument and performs the conversion based on certain properties.

Syntax: `\string.ToString(X):`

The argument X is a variable we want to convert. The variable has to be of arithmetical, boolean of GLPoint data type. For arithmetical variables we can specify by the property PlusString, which text sign has to be placed in front of the number, if it is positive. The text

sign has to be specified in between to vertical lines | |.

Syntax: `\string.PlusString = |+|;`

By default, no sign is written for positive numbers. For floating point numbers, the `FloagDigits` property allows to specify the number of decimal digits that are displayed.

Syntax: `\string.FloatDigits`

Apart from that, there are several different representations for floating point numbers. The property `FloatFormat` allows to choose between three different types.

Syntax:

```
\string.FloatFormat = string.FIXED_POINT;
\string.FloatFormat = string.EXP10;
\string.FloatFormat = string.MIXED;
```

If `FIXES_POINT` is chosen, the numbers are written out, i.e., the integer part is displayed followed by a dot and then followed by the number of decimal digits specified by `FloatDigits`. The value `EXP10` uses scientific representation of floating point numbers, i.e., only one digit before the dot is shown, followed by a number of decimal digits given by `FloatDigits`. This is followed by, e.g., e+02, which means that the previously number must be multiplied by 10 to the power 2. Finally, `MIXED` mixes the two previously mentioned variants. Here, `FloatDigits` also takes the digits left to the decimal dot into account. If `FloatDigits` is large enough to display the integer part fully, the remaining number of digits is written in standard decimal notation. Otherwise, the number is written in scientific representation.

Converting `Complex` variables we can choose the way the imaginary unit is displayed. The symbol or text can be modified using the property `IString`.

Syntax: `\string.IString`

An uppercase `I` is the default value.

The property `Remove0s` allows to modify the behavior of filling up decimal digits by zeros. If zeros should be shown, we set it to `true`, and to `false` otherwise.

Syntax: `\string.Remove0s`

Similar to `FloatFormat` for floating point numbers, the property `IntFormat` allows to change the representation of Integers.

Syntax:

```
\string.IntFormat = string.BINARY;
\string.IntFormat = string.OCTAL;
\string.IntFormat = string.DECIMAL;
\string.IntFormat = string.HEXADECIMAL;
```

Here, `BINARY` gives the number in binary representation (digits 0 and 1), `OCTAL` for octal representation (digits 0 to 7), `DECIMAL` for the standard decimal notation, and `HEXADECIMAL` gives the numbers in hexadecimal notation (digits 0 to 9 and A to F). In binary representation `PlusString` does not take effect, since the sign of the number is given by the first bit.

For `GLPoint` variables only the first three components are converted and the settings of `FloatFormat` apply. The three components are enclosed by parenthesis. Optionally, one can specify a `String` that is written out in front of the row vector. This string is given by the `GLPointString` property. By default `GLPointString = "GLPoint"`.

Syntax: `\string.GLPointString`

The conversion of truth values can be adjusted by giving the conversion of `true` and `false`, directly. The text substitutes are stored in the properties `TrueString` and `FalseString`.

Syntax:

```
\string.TrueString
\string.FalseString
```

To display special characters we can use their ANSI code in the `ANSIToString` function. Here, 256 symbols are listed and every symbol is mapped to an integer between 0 and 255.

Syntax: `\string.ANSIToString(code);`

Here `code` is an Integer between 0 an 255. The entries 32 to 127 of the ANSI table are identical to the ASCII table. The first 32 entries are command symbols. The corresponding arrangement of the symbols in the ANSI table depends on the configuration of your operating system. (see p. 46 in Section 3.4).

Finally, we deal with the converse direction, i.e., we want to convert variables of type `String` to numbers. Two commands are available depending on whether the result should be an `Integer` or `Real` variable: `ToInt` and `ToReal`.

Syntax:

```
\string.ToInt([radix,]s);
\string.ToInt([radix,]s,out);
```

The first variant returns the number, the second writes the resulting number to the variable `out`

and returns `true` if the conversion was successful. In all cases, `s` is a `String` variable which is converted to an `Integer`. The `String` must be of the following form:

`[w][s][0][x][ddd]`

where `[w]` stands for whitespace, `[s]` for sign (+ or -), `[0]` for an optional zero, `[x]` for a lowercase or uppercase `x` and `[ddd]` for the digits. All parts are optional, the conversion only needs the the ordering. If `radix` lies between 2 and 36, `s` is interpreted as a number given in the number format based on `radix`. If `radix` is 0 (default), the first character of `s` determines the number format according to Table 6.9.

First character	Second character	Interpretation
0	1 to 7	octal representation
0	`x` or `X`	hexadecimal representation
1 to 9		decimal representation

Tbl. 6.9: Determination of the base for `Integer` conversion

In particular, no dots or commas are allowed in the `String`. For conversion to floating point numbers, we use the `ToReal` command, instead.

Syntax:

```
\string.ToReal(s);
\string.ToReal(s,out);
```

The effect is similar to `ToInt`. The first variant returns the number, the second writes it to the variable `out` and returns `true` on successful conversion. This time, `s` has to be of the form:

`[w][s][ddd][.][ddd][fmt[s]ddd]`

where `[w]` marks whitespace, `[s]` the sign (+ or -), `[ddd]` the digits, and `[fmt]` stands for the exponential sign (`e` or `E`). Of course, `[.]` stands for the decimal dot. All elements are optional. Additional values for `s` are `+INF` and `-INF` for plus and minus infinity (see Section 3.2) and `NAN` for not-a-number.

6.3.2 Arrays

Arrays can be used to store several variables of the same data type in one place. Table 6.10 contains a list of data types that allow grouping within an array. Arrays constitute the SigMath counterpart to mathematical objects like vectors or matrices.

Definition

To define an array, we specify the name of the variable, the data type of the elements to store in it, its dimensionality and sizes in its dimensions. The general syntax reads as follows:

Array type	Data type of the elements
RealArray	Real
IntegerArray	Integer
ComplexArray	Complex
StringArray	String
BoolArray	Bool

Tbl. 6.10: Available array types in SigMath

Syntax:

```
\variable ArrayName = ArrayType(Number_Elements_1,...,Number_Elements_d);
```

Here, `ArrayName` is a valid name for a variable, see p. 75 for the rules to generate variable names. *ArrayType* is one of the array types listed in Table 6.10. Once this type is specified, all elements have to be of the corresponding data type. `Number_Elements_1,...,Number_Elements_d` is a list of nonnegative integers that specify the number of elements that can be stored in the corresponding dimensions of the array. If the list is empty, i.e., `d = 0`, the array cannot be used to store data. Nevertheless, the variable exists, and all methods can be called.

The total number of elements in an array, called *size* of the array is implicitly determined by the definition. It is given as the product of all numbers of elements in the dimensions.

$$\texttt{Size} = \texttt{Number\_Elements\_1} \times \texttt{Number\_Elements\_2} \times \cdots \times \texttt{Number\_Elements\_d}$$

The size of an array is zero as soon as one of the sizes of its dimensions is zero. Thus, arrays can have size zero although they have at least one dimension. The *layout* of an array is given by its dimensionality and the number of elements in each dimension. In SigMath there are no limitations on the size or layout of an array, i.e., up to restrictions of (physical) memory, arrays can take arbitrary "sizes". Samples of array definitions are given as follows:

```
\variable M = IntegerArray(3,4);
\variable N = RealArray(2,4,0);
```

This way, `M` is an `Integer`-valued 3×4 matrix, i.e., the first dimension contains 3, and the second dimension 4 elements. We can store $3 \times 4 = 12$ `Integer` variables in `M`. Its size is 12, its layout is 3×4. The array N has dimension 3 and layout $2 \times 4 \times 0$ but size 0 and thus cannot store any data.

In contrast to the data types we encountered before we do not have to initialize arrays, explicitely. To prevent inconsistencies, all elements of an array are set to the null element of the corresponding data type by default. For arithmetical data types the null element is `0` (`Integer`), `0.0` (`Real`), and `(0.0,0.0)` (`Complex`). For a `StringArray` it is given by the empty string `""` and for `Bool` the value `false`.

Since every element of an array has a unique position in the array, we can access the element by the index operator `()`. The position of the element is given as a d-tuple to the index operator. Thus, to obtain the value of the element of `M` at position `(i1,...,id)` we write

Syntax: `\M(i1,...,id)`

Indexing of arrays always starts with `0`. Thus, for a vector given by `V=IntegerArray(3)`, `V(0)` returns the value of its first element and `V(3)` does not exist. Calling this method returns a value in the form of a temporary variable, which has no identifier in the usual sense of a variable. Thus, single values have to be accessed and stored to variables (preferentially of the same data type) in order to be displayed using the `ValueOf` command. Two special elements of an array can also be easily accessed.

Syntax:
```
\A.front()
\A.back()
```

The `front` method returns the value of the first element of the array, the `back` method returns its last value. Here, ”first” and ”last” refer to the way of storing arrays in memory. They are stored sequentially in a connected block in memory and the interpretation of multiple dimensions i provided by using index-tuples. The order of storage is given by running the index of the last dimension first, and so on. We illustrate the principle on a $2 \times 2 \times 2$ array.

$$\texttt{A} = a_{000}\ a_{001}\ a_{010}\ a_{011}\ a_{100}\ a_{101}\ a_{110}\ a_{111}$$

For example, a two-dimensional matrix is stored by concatenating its rows. This way, `A.front()` returns the value of a_{000} and `A.back()` the value of a_{111}.

Operators on arrays

Arithmetical operators (see Section 6.2.4) are also defined for arrays. Besides data type compatibility, the layouts of the operands have to match. That means, two arrays can only be added or subtracted if the number of elements equal. The resulting array has the same dimensions as the first summand. The operator `*` results in a matrix multiplication and is thus available for two-dimensional arrays with the following layout `r`×`n * n`×`d`. This command results in a `r`×`d`-matrix. Automatic type conversion is performed as for elementary data types (see Table 6.7). Taking powers or division of arrays is not implemented, but scalar multiplication is performed elementwise.

Two arrays are equal if they have the same dimension, size, type and match in all elements. This definition excludes type conversion in connection with comparisons. Whereas `2 == 2.0` returns `true` for scalar values, the `IntegerArray (2,2)` and the `RealArray (2.0,2.0)` are not equal. Besides `==` and `!=` no other comparison operators are available for arrays.

Logical comparisons and bitwise operators are not admissible. The increment operators `++` and `--` do not apply to single array components. For write-access on an array element one has to put the corresponding element on the left hand side of an assignment. In case of arithmetical arrays also compound assignments are admissible. Writing `\A = 3` results in an array whose elements all equal `3`.

Properties of arrays

Given an array `A` we can determine its overall size or the size of one dimension by using the following methods:

Syntax:
```
\A.size()
\A.size(i)
```

The special form `size(i)` returns the number of elements in the `i`th dimension. Note that the method `size` for arrays has a different behavior than the `Size` method in the class `string`. For arrays, it is read-only. To change the layout of an array one has to use the `Resize` method.

The dimensionality of an array can be obtained by the `Dim` method:

Syntax: `\A.Dim`

To detect if the array has size 0 one can either write `A.size()==0` or use the `empty` property. If `A.size()==0` it returns `true`.

Syntax: `\A.empty`

Every array has a version number, that is initialized by `0` on definition. Every modification of the array increases the version of the array. Using `Version` we can obtain the current version number. To increase the version number manually, one uses `SetVersion`.

Syntax:
```
\A.Version
\A.SetVersion();
```

Since plot objects in SigMath react to the change of version numbers (see Section 4.1.3) one can use the second method to force SigMath to redraw the content of an array.

Methods to fill arrays

The basic version to fill the elements of an array is given by writing for every element its own assignment using the index operator. More convenient is the usage of the method `FillFromString`:

Syntax: `\A.FillFromString(s);`

Here `s` is a variable of type `String` and contains the data that will be stored in the array. The storage pattern of the resulting array heavily depends on the layout of the array `A`. Let us illustrate this point with the help of some examples.

```
\variable R = RealArray(0);
\variable C = ComplexArray(0);
\variable I = IntegerArray(0,0);
\variable S = StringArray(0);
\R.FillFromString("15 2 3 4");
\C.FillFromString("15 2 3 4");
\I.FillFromString("15 2
                   3 4");
\S.FillFromString("15 2
                   3 4");
```

`R` gives an array of real numbers of dimension 1. The size of the array does not take effect in the `FillFromString` method since it is automatically set depending on the input argument and overwrites all previous size definitions. The blanks serve as separators, i.e., the string has to be interpreted as a list of four constants of type `Real`.

Next we define `C` as a `ComplexArray` of dimension 1. Since variables of type `Complex` consist of real and imaginary part, we get a different result on the same argument. Now, `C` is a vector of length 2 with the elements `15 + 2*I` and `3 + 4*I`. Missing elements are not automatically filled up. Thus, one has to provide a string consisting of an even number of numbers.

Besides blanks also line breaks serve as separators. Blanks are used to indicate the next element of the same row, newlines indicate a jump to the next column of the array. In the given example, we thus obtain a 2×2 matrix of `Integers` with `I(0,0)=15, I(0,1)=2, I(1,0)=3, I(1,1)=4`.

If we try to fill a `StringArray` using the `FillFromString` method, the blank is not considered as a separator since `String` variables may also contain blanks. Thus, only a new line serves as separator for elements. Thus, only 1-dimensional `StringArray`s can be filled with this method.

For two-dimensional arrays, i.e., matrices, we can use `SetBlock` and `GetBlock` to write submatrices or extract submatrices.

Syntax:

```
\A.GetBlock(firstRow,lastRow,firstCol,lastCol,B)
\A.SetBlock(firstRow,lastRow,firstCol,lastCol,B);
```

The first four arguments specify the index of the first row and first column and the last row and last column of the submatrix under consideration. Using `GetBlock` the elements are assigned to the variable `B`, using `SetBlock` the elements are filled up with the entries of `B`.

If we want to save of load array variables from or to files, we use `Save` and `Load`.

Syntax:
```
\A.Save(filename);
\A.Load(filename);
```

Here, either the content of `A` is written to the file with name `filename` or `A` is filled with the content of `filename`. For the `Save` method, the file name extension controls if the array is stored as an image (`.jpg` or `.bmp`) or as text. With `Load` one can load an array which was previously stored by a `Save` command.

To display an array in closed form, SigMath provides a method to convert arrays to strings.

Syntax: `\A.ToString(s);`

`ToString` stores in the variable `s` of type `String` a list of elements separated by blanks and ordered in the way they are stored. For matrices the line break is depicted. For higher-dimensional arrays the structure is lost and the resulting string contains only one single line.

Within the `document` environment, `ArrayBlock` is a comfortable alternative to `ToString`.

Syntax: `\ArrayBlock(name);`

The argument `name` is a `String` that gives the name of the array. The following lines of code display the array `A` on the screen according to its layout. For example, a 2×3 matrix is given in 2 lines and 3 columns.

```
\variable A = IntegerArray(2,3);
\begin{array}
 \ArrayBlock("A");
\end{array}
```

This type of display can only be used for one- or two-dimensional arrays, i.e., vectors and matrices. An array of higher dimension must be rearranged using the `Resize` method before displaying it with `ArrayBlock`. For the converse task of writing a one or two-dimensional table (as an `array` environment, see p. 35) to an `array` variable one uses the `InputArrayBlock` command which has been introduced in Section 3.3.

Methods to manipulate arrays

After filling the array and showing it, we want to introduce means to change the values of an array. All methods work on the corresponding array, directly. That means, previous layout may be overwritten and get lost. Also note that some of the following methods are only available for one- or two-dimensional arrays. First, we can rearrange an already existing array to obtain some higher-dimensional array.

Syntax:
```
\A.Resize(n0,...,nd);
\A.ResizeInit(n0,...,nd,defaultValue);
```

The first line transforms `A` to an array of layout `n0`$\times\ldots\times$`nd`. If this increases the size of the array, missing values are filled up with null elements of the corresponding data type. The second variant can be used to change the default behavior. All missing values are automatically filled up by `defaultValue`.

```
\variable I = IntegerArray(0);
\I.FillFromString("1 2 3 4 5 6 7 8");
\I.Resize(2,2,2);
```

In this example, we generate a one-dimensional array, i.e., a vector, of length 8 and rearrange it afterwards to an three-dimensional array with layout $2 \times 2 \times 2$. The rearrangement follows the sequential storage pattern we introduced on p. 94. The result of our transformation reads as `I(0,0,0) = 1`, `I(0,0,1) = 2`, `I(0,1,0) = 3`, `I(0,1,1) = 4`, `I(1,0,0) = 5`, `I(1,0,1) = 6`, `I(1,1,0) = 7`, and `I(1,1,1) = 8`.

For two-dimensional arrays we can interchange the meaning of columns and rows, i.e., we transpose the matrix by `Transpose`.

Syntax: `\A.Transpose();`

Traversing the sequential storage scheme of an array backwards, we revert the array. This can be done by using `reverse`.

Syntax: `\A.reverse;`

For one-dimensional arrays, we can insert a new data value at a certain position. This is performed by `insert`.

Syntax: `\A.insert(j,value);`

The index `j` must be `Integer` and lie between 0 and `A.size()-1`. The data type of `value` has

to match the data type of the array. To append a value at the end of a vector, we can use `push_back` method.

Syntax: `\A.push_back(value);`

As before, `value` has to match the data type of the array. The method `pop_back` allows to delete the last entry of a vector.

Syntax: `\A.pop_back();`

To delete the first entry of a vector, we can mirror the vector by `reverse`, then execute `pop_back` to delete the first entry (which has become the last of the reverted vector) and then mirror back.

Further mathematical methods on arrays

For arrays of arithmetical data types we can use further methods that implement mathematical functions on the arrays.

With `Mean` we can compute the (arithmetical) mean of all elements of the array and `Sum` computes the sum all elements.

Syntax:

```
\A.Mean
\A.Sum
```

The methods returns the mean and sum of the array which can be stored to another variable.

6.3.3 GLPoint

The data type `GLPoint` is comparable to a one-dimensional `RealArray` with four elements. As indicated by the name, this data type may be used to store information on points in three-dimensional space. This data type is especially useful in connection with OpenGL commands and simplifies syntax. In this section we focus on the general description and usage of this data type. Its connection with OpenGL programming is postponed to Chapter 7.

Definition

The following example shows the correct way to define a variable of type `GLPoint`. The arguments are interpreted as `x`, `y`, `z`, and `w` coordinates of a point. For the interpretation of the last coordinate we refer to the chapter 7 on OpenGL.

Syntax:

```
\variable name = GLPoint(x,y,z,w);
\variable name = GLPoint(x,y,z);
\variable name = GLPoint(x,y);
```

```
\variable name = GLPoint(x);
\variable name = GLPoint();
```

Missing values in the first three coordinates are automatically set to `0.0`, a missing fourth component `w` is set to `1`.

Using the dot notation we can access every single coordinate of a `GLPoint` variable.

Syntax:

```
\P.x;
\P.y;
\P.z;
\P.w;
```

Here we assume `P` to be a variable of type `GLPoint`. Then `P.x` returns the first, `P.y` the second, `P.z` the third, and `P.w` the last coordinate of `P`.

The command `Input/ValueOf` allow to ways to access its values. We can either specify a single coordinate or the whole point. The following example illustrates both approaches.

```
\variable P = GLPoint(1,2,3);
\variable i = 1;
\ValueOf("P", i);
\ValueOf("P");
```

In the first variant we display the value of the `i`th coordinate, only and interactive modification is possible. `i` may only take values in `1` and `4`. The second variant displays `GLPoint(P.x,P.y,P.z)` on the screen not allowing any manipulation.

Methods

For `GLPoint` variables arithmetical operators are defined. For addition, subtraction, and compound assignments the same rules as for arrays apply. The multiplication operator `*` computes the inner product of the two vectors, though. Note that the fourth component is ignored in this computations! Let us give some examples for arithmetical operators with `GLPoint` variables.

```
\variable P = GLPoint(1,2);
\variable Q = GLPoint(2,4,6,8);
\variable R = GLPoint(3,5,7,9);
\variable S = P + Q;
\variable T = R * S;
```

The type of the variable `T` is detected on initialization as `GLPoint`. Since `P.y = 0` and the fourth coordinate is ignored, the value of `S` is given by `S = (1+2,2+4,0+6,1) = (3,6,6,1)`.

The multiplication returns the value of the inner product, i.e., `T = 2*3 + 4*5 + 6*7 = 88`.

The comparison operator `==` is defined as for arrays. Two `GLPoint` variables are equal if they equal in all elements. Type conversion do not apply here, since elements of a `GLPoint` vector are always of data type `Real`. The fourth component is not ignored! In addition to the class `array` the comparison operators `<=`, `<`, `>=`, and `>` are defined for `GLPoint` variables elementwise.

Using the method `Length` computes the (Euclidean) length of a vector, `SquaredLength` computes is square.

Syntax:

```
\P.Length
\P.SquaredLength
```

Here, length is the Euclidean length of the three-dimensional vector obtained by ignoring the fourth coordinate. The returned value has data type `Real`. By this method, the length of a `GLPoint` variable can be adjusted, too. If we assign a new length to a `GLPoint` variable `P` by `P.Length = x` the vector is rescaled (preserving its direction) to the new length `x`. The second method `SquaredLength` only allows read-access.

To determine the angle between two vectors, we can use the methods `AngleDegree` and `AngleRadian`. The first method returns the angle in degrees, the second in arc length.

Syntax:

```
\P.AngleDegree(q)
\P.AngleRadian(q)
```

Both methods need an argument `q` of type `GLPoint` and do not change `P`. If one of the vectors is the zero-vector, both methods return `0`.

The method `CrossProd` allows to save the vector cross product of two vectors to the calling variable.

Syntax: `\P.CrossProd(a,b);`

Here, `a` and `b` are two `GLPoint` variables. The vector `P` is then orthogonal to `a` and `b` and its length is determined by the area of the parallelogram which is spanned by `a` and `b`. As before, the fourth coordinate is ignored in the computations and in the resulting vector `P.w` equals `1`.

The remaining methods serve as interfaces to OpenGL functions and are thus treated in Chapter 7.

6.3.4 Container

Container is a general expression for complex data structures which can be used to organize and store data elements. There are different types of data storage strategies which are implemented as different types of containers. Each strategy has its advantages and disadvantages. The choice of the correct container depends on the later usage.

Although the implementation of arrays and containers in SigMath differs, we use arrays to introduce the fundamental concepts connected with containers. Arrays allow fast access to single data elements using the index operator. If we do not know the exact position of a single element in an array, though, the data structure does not allow for efficient search strategies. Here, a sorted data structure increases efficiency. Such data structures are, e.g., realized by containers of type `Set`. As a complex data structure the class `cont` provides methods for containers. There exist methods for all types of containers and specialized methods only applicable to containers of a certain type.

Definition

All implemented types of containers can be found in the following syntax to generate empty containers.

Syntax:

```
\variable S  = Set();
\variable MS = Multiset();
\variable M  = Map();
\variable MM = Multimap();
\variable L  = List();
```

To use container variables in functions or as function arguments or for operators we have to specify the data type of a container (see Section 6.5).

Syntax:

```
\variable "cont" S  = Set();
\variable "cont" MS = Multiset();
\variable "cont" M  = Map();
\variable "cont" MM = Multimap();
\variable "cont" L  = List();
```

We give a list of the implemented container types and a short description in Table 6.11.
All introduced container classes share the following properties:

1. Every container may consist of an arbitrary number of data elements. In contrast to arrays we do not have to specify neither size, nor data type beforehand. This guarantees a high level of flexibility and storage efficiency. The only limitation on mixing data types is that

Type	Description
`Set`	A `Set` contains a sorted set of data elements. Every element must be unique. This restriction increases the complexity for inserting new elements. On the other hand, fast access to data elements is provided.
`MultiSet`	In addition to `Set` the data structure `Multiset` allows elements to appear multiple times.
`Map`	A `Map` contains elements consisting of its value and of a key. The elements are sorted with respect to their key. Every key must be unique. This way of storing data allows fast access of the elements of the container.
`MultiMap`	In addition to `Map` the data structure `MultiMap` allows keys to appear multiple times.
`List`	A `List` stores its elements unsorted, but in a determined order.This data structure is optimized for fast insertion and deletion at arbitrary locations.

Tbl. 6.11: Container types in SigMath

they have to be comparable. That means, the comparison operators `==` and `<` must be applicable for any two elements of a container.

2. If we insert a data element `A` into a container only its value is inserted. Thus, altering the value of `A` does not affect the content of the container.
3. The elements of a container are ordered. The way they are sorted depends on the type of container. The type `Set` guarantees an ascending list, elements of `List` are appended to the end of the list.
4. In contrast to arrays operators on containers are not secure. That is, the programmer of the code has to make sure, that all operands of the operators lie within admissible ranges.

Access to elements in a container is only possible by so called *iterators*. Those methods guarantee that the data structure is always in a well-defined state.

Internally, an iterator is a pointer which addresses the content of a container depending on its type. Iterators allow to navigate to certain positions within the container and allow to read data or insert new elements. The iterator represents a position within the container and can be used as an argument for a method. Let us have a look on an example.

Example 6.1 We define the container variable `S`. The container `Set` has the property that its elements are stored in sorted order. Every time we include elements by `insert()` to `S`, the correct position is determined internally. The method `ToString()` is used to convert the given container variable to a `String` and returns the result. This function uses the iterator `i` as a loop variable of the `for` loop and thus runs through the set `S`. The method `begin()` returns the iterator on the first element of the container. This is set to the next element by applying the

increment operator ++ until we reach the end of the container. The last element of a container is addressed by `end()`. In between, the method `ValueAt()` which needs the iterator as its argument reads the value at the corresponding position. Finally, the compound assignment += generates the resulting `String`.

Code:

```
\function ToString("cont" S) =
{
  variable res = "";
  for (variable i = S.begin(); i != S.end(); ++i)
    res += " " + S.ValueAt(i);
  return res;
}

\begin{document}
  \variable S = Set();
  \S.insert("This is");
  \S.insert("the way");
  \S.insert("to sort");
  \S.insert("Strings");
  \S.insert("!");

  \variable Display = ToString(S);
  \ValueOf("Display");
\end{document}
```

Methods for containers

In this section we introduce some of the elementary methods provided by the class `cont` for containers.

To detect the number of elements in a container, SigMath provides the method `size`. If one wants to decide if a container is empty, we suggest to use `empty`. The result is of type `Bool` and has the value `true` if there are no elements in the container. The method `clear` deletes all elements of a container and leaves the container empty. The method returns the data type `Void`.

Syntax:

```
\S.size();
\S.clear();
\S.empty();
```

Example 6.2 Let us have a look at the effect of these methods.

```
\variable S = Set();
\S.insert(5);
\variable i = S.size();
\S.clear();
\S.empty();
```

The first line generates a new container `S` of type `Set`. Then we add the element `5` using `insert`. The query `S.size` returns 1 and assigns this value to the `Integer` variable `i`. In the following `S.clear` empties the container which results in the answer `true` in the last line `S.empty`.

For the user all iterators have produce the same effect on all types of containers. Internally, every container type implements its own iterator method. The methods `begin` and `end` provide iterators which point to the beginning and the end of a container. For the resulting value of the method the actual type of the container does not take effect. Only the way the result is generated depends on the type.

The methods `pop_front` and `pop_back` allow to delete the first and last element of a container, respectively. The methods `push_front` and `push_back` allow to move the data element from its current position given by the iterator to the beginning or to the end of the container.

We can compare container variables using `==` and `!=`. Two containers are equal, if they contain the same elements in the same order. Otherwise the result of a comparison is `false`.

The assignment operator `=` is defined for container variables, too. The data types of the operands have to match, i.e., no automatic type conversion is performed. Afterwards, both containers equal the right operand. Using `swap` we can exchange the content of two containers of the same type. After issuing `A.swap(B)` the variable `A` contains the contents of `B` and vice versa.

6.3.5 Bitmaps

6.3.6 Text editors

6.4 Control structures

By default all commands in a SigMath document are executed in the order they are written in the document. This section introduces control structures that allow to break out of this sequential execution. We distinguish two structures: branches or loops. Using a branch allows to execute different commands depending on the value of a condition. Thus, there are different ways to traverse the document. A loop can be used to repeat certain lines of code. The number of repetitions depends on a condition which has to be specified by the programmer.

6.4.1 Branches

In SigMath we distinguish between *simple* branches which are realized by `if-else` structures and *multiple* branches which are realized by `switch-case` structures. We start with simple branches.

The simple branch

Simple branches allow to execute different code depending on the value of a `Bool` expression called *condition*. The branch is realized by using the `if` command followed by a condition. If the condition evaluates to `true` the following block of commands is executed, otherwise SigMath looks for a command block beginning with `else`. If now `else` block is specified SigMath continues with the execution of code following the `if` block.

Syntax:

```
if (condition)
{
 commands
}
else
{
 commands
}
```

In case that the `if` or `else` block consists of one command, only, the brackets may be omitted. Simple branches can be nested. In that case, every `if` block has to be accompanied by an associated `else` block. The connection is determined by the structure of the nesting. Let us consider an example.

Example 6.3 A simple `if-else` structure can be used to display the reciprocal of a real number `y`. We use `if` to compute the quotient of two numbers.

1. Simple `if-else` structure:
```
if (y == 0.0)
 result = INF;
else
 result = 1/y;
```

2. Nested `if-else` structure:
```
if (y == 0.0)
{
 if (x == 0.0)
   result = NAN;
 else
```

```
   result = INF;
}
else
  result = x/y;
```

3. Computing the quotient without nested branches:

```
if (y == 0.0 && x == 0)
 result = NAN;
if (y == 0.0 && x != 0)
 result = INF;
if (y != 0.0)
 result = x/y;
```

Since the cases `0/0` and `x/0` require special consideration we use nested branches in the second example. This requires two comparisons in case that `y==0` is true. The last variant of computing the quotient has five comparisons, since every condition has to be checked, separately.

The condition must be a boolean expression, i.e., on evaluation it returns a value of data type `Bool`. If not, compilation of the document fails and an error message is displayed.

The evaluation of complex expressions like

```
A || x == ++y;
```

needs further consideration. Some programming languages use *lazy evaluation*, i.e., as soon as the first condition `A` evaluates to `true` the whole expression is set to `true`. In contrast, SigMath evaluates all parts of the expression. This way, all subexpressions are investigated which prevents some undesired effects.

SigMath also provides the command `IgnoreIf` that allows to ignore code if a certain condition is satisfied.

Syntax:

```
\IgnoreIf(condition);
 code
\endif
```

If the evaluation of the condition returns `true` the whole code segment within `IgnoreIf` and `endif` is ignored.

Multiple branches

The `if-else` structure only allows boolean expressions as conditions. If one wants to check more than one condition at the same time, we suggest to use the `switch-case` structure. It

performs a distinction of cases. One argument is evaluated by the following `case` commands. If one condition is satisfied, the corresponding code segment is executed. To exit the execution of the `switch-case` structure one can put a `break` command at the end of a command block. This terminates the execution of the *whole* `switch-case` block. Alternatively, more than one case may be satisfied and executed. In analogy to `else` there is an optional `default` case available. It is always executed if none of the alternatives is satisfied. The `default` case should always be at the last position. If we close all case blocks by `break` the `default` case is executed if no other case is executed. The syntax for a `switch-case` structure reads as follows:

Syntax:

```
switch (expression)
{
 case A1:
   commands;
   break;
 case A2:
   commands;
   break;
 ⋮
 default:
   commands;
   break;
}
```

Example 6.4 In this example we want to check if `x+5` equals 3, 10 or none of the two cases. We store the result of this comparison in a text variable (in words).

```
switch (x + 5)
{
 case 3 :
   Display += "The result is 3";
 case 10:
   Display += "The result is 10"; break;
 default:
   Display += "The result is not 10"; break;
}
```

First, the simple expression `x + 5` is evaluated. If the result is `3`, the first case applies. If `x + 5 == 10` is satisfied, the second case applies and we terminate the `switch` block by a `break`. In this example the first case is missing the `break` command. If the first case applies, the default case is evaluated, too. In the case of `x + 5 == 10` we do not reach `default`.

Another interesting effect is illustrated by the following example.

```
switch (x)
{
 case 1:  ++x;
 case 2:  ++x;
 case 3:  ++x;
}
```

What is the value of `x` after executing the `switch-case` structure if the initial value is `1`? The correct answer is `4`, since by incrementing `x` all cases apply and no `break` command is used.

Multiple branches can be nested as simple branches.

SigMath does not restrict the type of expressions used in the `switch` condition. The comparisons in the `case` statements are performed using automatic Al type conversions as indicated on p. 81. Thus, `5 == 5.0` is true. If no type conversion is possible, the result is automatically set to `false`. Thus, comparisons like `5 == "Text"` are syntactically correct. This feature turns `switch-case` structures into a very flexible programming tool.

Example 6.5 We already encountered the `switch-case` structure in connection with function calls by interactive elements (see Example 3.3). A function `f` may be called by different `DoString` commands on different pages of a document (see Section 3.1). Depending on the calling element, we can execute different code blocks in the function. The last example illustrates the usage of `switch-case` structures in functions. In general, the function `f` has, e.g., the following structure.

```
\function f() =
   {
    switch (SelectedDo)
    {
      case ident1:
           commands
      case ident2:
           (other) commands
      default:
           commands
    }
   }
```

6.4.2 Loops

Loops can be used to execute certain code blocks several times. If the number of repetitions is known before we use counting loops, whereas `while` loops allow a little bit more flexibility.

A loop structure consists of two parts: *loop header* includes the type of loop, the number of iterations or conditions on the number of repetitions, whereas the *loop body* consists of code that is executed depending on the loop header commands several times.

Counting loops

Counting loops or `for` loops allow to repeat a certain code segment a "fixed" number of times.

Syntax:

```
for (initialization; condition; reinitialization)
{commands}
```

A vital part of any `for` loop is the *loop* or *counting* variable. It is initialized in the loop header, i.e., before the loop body is executed the first time. All commands in the `initialization` part are only executed once. The `condition` typically contains the counting variable. Depending on its truth value, the loop body is executed. In particular, it may happened that the loop body is not executed at all. After the last command in the loop body, `reinitialization` is performed and the condition is checked with the new values of the variables. Thus, `reinitialization` is executed before entering the loop body a second time and then every time before reentering the loop. As soon as the condition is not satisfied any more, repetition stops and execution resumes at the next command after the the loop.

Example 6.6 Let us have a look at a very simple example. The following code calculates the sum and product of the first four integers.

```
for (variable i = 1; i < 5; ++i)
{
 Sum   += i;
 Product *= i;
}
```

The counting variable `i` is defined in the loop header and initialized by the value `1`. Its scope is limited to the loop. This memory-saving way should be used any time the counting variable is not used outside of the loop. Together with the reinitialization expression the condition leads to four repetitions of the loop body. Reinitialization may also contain more complex updating expressions.

```
for (variable i = 2; i < 10; i = i ^ 2)
 Sum += i;
```

This time, the loop body is only executed twice, since `i` has the value `16` before entering the loop a third time. As with `if-else` structures, we can omit the enclosing brackets if the loop body consists of one line, only.

The loop header. The `initialization` part of the loop header may have two forms. Either we define a (new) counting variable in the loop header which then has restricted scope, or we use a previously defined variable. In the second case the counting variable remains visible after the loop and its value is the resulting value of the last reinitialization. Within the loop body we have unlimited read and write access to the counting variable. While read access does not influence the behavior of the loop, write access on the counting variable may result in uncontrollable behavior. One might even get stuck in an infinite loop. Note that reinitialization must not be closed by a semicolon. SigMath does not support the usage of more than one counting variable within one `for` loop and the condition as well as the reinitialization must not consist of more than one command.

Aborting the loop By default execution of the loop terminates as soon as the condition in the loop header is no longer satisfied. To abort the loop one can add a `break` command. Depending on another condition inside the loop body one can stop further repetitions. In case of a `break` SigMath resumes execution at the first command after the loop. If one wants to stop the execution of the current repetition, only, one uses `continue`. Then the remaining commands in the loop body are skipped and a new repetition starts.

Example 6.7 Let us consider the following question. Given a small non-negative real number ε, e.g., $\varepsilon = 0.0264$, we want to compute the smallest integer $i \in [1;100)$ such that $1/i < \varepsilon$. One (naive) idea is to use a `for` loop with counting variable `i`.

Code:

```
\variable epsilon = 0.0264;
\variable i = 1;
for (i = 1; i < 100; ++i)
{
 if (1.0/i > epsilon) continue;
 else Display += "i is" + \ValueOf("i");
 break;
}
```

As long as `i` is not large enough, we directly restart the repetition using `continue`. This is done until either `i==100` or we the condition is satisfied. In this case, the value of `i` is displayed and the `for` loop is aborted by `break`.

While loops

Another possibility to repeat the execution of a code segment is given by `while` loops. The loop header contains the `while` command and a condition. Initialization has to be performed beforehand and the reinitialization is part of the loop body. Thus, there are basically no limitations. We can, e.g., use more than one loop variable.

Syntax:
```
initialization;
while (condition)
{
 ...
 reinitialization;
 ...
}
```

The `while` loop is a general form of a loop structure. As before `break` and `continue` can be used to influence the execution of the loop body.

6.5 Functions

An efficient strategy to solve complex problems is to identify smaller subproblems that are easier to solve. One then tries to solve the subproblems, first and can try to find a solution for the overall problem from the solutions of the subproblems. In this way, the solutions of smaller subproblems may be used for different topics that contain the same subproblems.

In terms of programming languages, such subproblems or subtasks are represented by *functions*. They act as self-contained programs and can handle subproblems independently of the rest of the program. They help to improve readability and reuseability of code. Functions act like machines in a large factory. They ask for input, then work on the input in a previously determined fashion and produce a result which is returned or moved to another function.

In SigMath functions can be defined everywhere in the document. Functions might be nested, i.e., defined within other functions. In this case, the inner function is only available within the outer function (see Section 6.2.2).

The rules for names of variables (see p. 75) apply for names of functions, too. Functions come with their own scope: variables, which are defined within a function are local variables of the function. Thus, they may hide global variables.

To use a function one does not have to access its source code but only needs to know the class or document it is defined in and its interface: its name and a list of arguments given in parenthesis. Finally, we need to know the data type of its return value to use the function properly.

Function definition

Before we can use a function, we have to define the function using the following syntax:

Syntax:
```
function functionName (argument1, argument2, ...)  =
```

```
{
 commands
 return ReturnValue;
 commands
}
```

The first line is the *function header*. The reserved word `function` begins the definition of a function. Its name allows us to call the function at any position of the document. The name of a function within one block has to be unique even if the functions would have different lists of arguments. Only the name of the function is used to identify the function. SigMath will execute the function if the given list of arguments in the function call matches the list of arguments given in the definition. The list of arguments may contain zero or more arguments. `argument1, argument2, ...` are identifiers according to the naming conventions of variables. Within the function the arguments can be used as normal variables. They are initialized on function call according to the assigned values. Additional definitions of the arguments are not possible. Different arguments must have different identifiers.

The part in the brackets after `=` is called the *function body*. All commands in the function body are executed on the function is called until we reach the first `return` command. Thus, `return` has two effects: the variable or constant following it is the returning value of the function and `return` closes the function. All commands after the first `return` will not be executed. Nevertheless, a function might contain more than one `return` command, e.g., within `if` or `switch` commands. Since the data type of the return value need not be specified, functions provide a high level of flexibility.

Recursive function calls are not (yet) implemented in SigMath. Thus, a function cannot call itself. This does not exclude expressions of the form `f(f(x))`, since this is in fact an iteration. The return value of the "inner" function call is the input value of the "outer" call, as long as the return value matches the specifications of the arguments in the definition of `f`.

Function arguments

Let us make some comments on the usage of function arguments. Typically, the data type of an argument need not be specified. Thus, arguments having the same name but different data types may appear. If one wants to use certain methods for complex data types on the arguments in the function body it is advisable to specify the data type of the argument in quotation marks in front of the identifier in the function header. In particular, we recommend this method for the classes `array`, `cont`, `glpoint`, and `string` that implement complex data types (see Section 6.1 and Table 6.1).

Example 6.8 Let us give examples for function definitions. The first function `f` has two arguments `c` and `A` and returns the value of the expression `c^2 * A`. The argument `A` may be a primitive data type or an array. All commands within the function body can handle both cases.

`Sum("array" A)` is built to sum up all elements of an array. Since the function body uses the method `A.size()` and the index operator `A(i)` we specify in the function header the class `array`.

Code: (complexArguments.txt)

```
\function f(c,A) = { return c ^ 2 * A; }

\function Sum("array" A) =
{
 variable Result = 0;
 for (variable i = 0; i < A.size(); ++i)
   Result += A(i);
   return Result;
}
```

For arguments in a function call SigMath uses *call-by-reference*. That means, that the values of the arguments are not stored in the memory separately. Thus, changing the value of an argument in the function body changes the value of the variable used in the function call! The following example shows the effect of the expression `f(x)` on the value of `x` although `x` does not appear on the left hand side of an assignment.

```
\function f(a) = { a = 10; }

\begin{document}
 \variable x = 4;
 \f(x);
 \ValueOf("x");
\end{document}
```

Since write access on arguments may affect global variables, one should be careful about that. If the function call involves constants or other expressions, their value is stored temporarily during function evaluation and is no longer accessible after the execution.

6.6 Multithreading

In order optimize code one should think about certain subtasks than can be run simultaneously (concurrently). The class `thread` provides some commands to define and run independent *threads*. We can either execute different functions or several instances of one function in concurrent threads. In order to associate a function to a thread we write `thread.BeginThread`.

Syntax: `\thread.BeginThread(functionName);`

Here `functionName` gives the name of a function without arguments. The associated thread stops as soon as the execution of the function is terminated. All threads are registered within

the document. On closing of the document all threads are terminated and an error message is returned. This may result in inconsistencies. Thus, we recommend to make sure that all threads are terminated in a regular way.

Example 6.9 In the following example two threads are active (SigMath itself and the thread associated with the function `Count`.

```
\variable Stop = false;
\variable i    = 0;
\function Count() =
{
 while (!Stop)
   ++i;
}

\thread.BeginThread("Count");

\function CheckThread() =
{
 if (Stop)
   return true;
 MessageBoxOK(false, "Click 'Stop' to finish all threads.");
 return false;
}

\begin{document}
\OnDocClose("CheckThread");
 i    =\ValueOf("i");
 Stop =\Input("Stop");
\end{document}
```

The preamble of the example starts a thread that keeps on incrementing the value of the variable `i` which is displayed. The function runs as long as the Boolean variable `Stop` has the value `false`. If one wants to close the document, `CheckThread` is started. This function checks the value of `Stop`. If `Stop=true` the function `Count` exits, the thread terminates and the document can be closed.

Running user-defined functions in threads may result in inconsistent behavior since they use local static variables and arguments. To prevent different threads from using the same variables or functions at the same time we can use a so-called *critical section*. Within such a section only one thread can access certain variables at a time. We define a critical section by `thread.InitializeCriticalSection`, enter it by `thread.EnterCriticalSection`, and close it by `thread.LeaveCriticalSection`.

Syntax:

```
\thread.InitializeCriticalSection();
\thread.EnterCriticalSection(identifier);
\thread.LeaveCriticalSection(identifier);
```

Here `thread.InitializeCriticalSection()` returns an index of type `Integer`, which uniquely determined the critical section. The argument `identifier` is then the index of the previously defined critical section.

Example 6.10 Let us consider two functions that act on the same array.

```
\variable Stop = false;
\variable A    = RealArray(2,20);
\variable CS_ID = thread.InitializeCriticalSection();

\function Shuffle() =
{
 while (!Stop)
 {
   if (mod(syst.ProgTime,2)<1)
     A.Resize(2,40);
   else
     A.Resize(2,1);
 }
}

\thread.BeginThread("Shuffle");

\begin{document}
 \thread.EnterCriticalSection(CS_ID);
 \begin{array}
   \ArrayBlock("A");
 \end{array}
 \thread.LeaveCriticalSection(CS_ID);
\end{document}
```

The critical section takes care that the function `Shuffle` changes the layout of the array `A` while `ArrayBlock` displays the value of the array.

Example 6.11 As a last example we want to run one function `SimpleTask` several times and simultaneously. To do this we have to create several copies of the function (and maybe of certain variables) in a template.

```
\variable results#i = 0;
\function SimpleTask#i() = { ...}
```

Here `#i` acts as wildcard character that will be replaced by a number later on. We store this template to a file called `TemplateSimpleTaks`. To create several copies in our document we add the following lines

```
\variable Template = "";
\Template.Load(TemplateSimpleTask);
\variable RegID = CurrentRegID;
\for (variable i = 1; i <= NumberCopies; ++i)
{
 variable "string" Temp= "";
 Temp = Template;
 Temp.Replace("#i", string.ToString(i));
 ExecuteString(RegID, Temp);
}
```

In this way we created `NumberCopies` versions of the function `SimpleTask` and of the variable `results`. Now we can run each copy in a different thread:

```
for (variable i = 1; i <= NumberCopies; ++i)
    thread.BeginThread("SimpleTask"+ string.ToString(i));
```

6.7 Debugging

More complex programming tasks involve testing of the function and, unfortunately, trying to find errors. SigMath's editor can be used as a debugging tool. In this section we introduce two general types of errors and the debugging tools in SigMath.

Basically there are two types of errors, *syntax* and *semantic errors*. Examples for syntax errors are missing parenthesis or typing errors in reserved words. If we try to compile a document containing syntax errors using the built-in text editor of SigMath, compilation will abort, show an error message and SigMath display the line that contains the error. If the error is contained in an external document, the document is opened and the corresponding line will be shown. If the error occurs within one environment, the line ending the environment is indicated. Syntax errors are easy to fix, most of the time.

On a semantic error, the document compiles since the correct syntax is used, but the program does not act as it was intended to do. Reasons may be unwanted rounding issues due to the usage of wrong data types or logical errors like improper conditions in control structures. In

order to ease the search for semantic errors SigMath provides the `BreakPoint` function.

One vital tool to find semantic errors is to monitor values of variables. Since SigMath is updating the document in every frame, this might be a hard task. Here, `BreakPoint` can be used. It interrupts the execution of the program and returns the values of the active variables in the current environment. On a mouse click, the execution is resumed until the next `BreakPoint` is reached. After the last `BreakPoint` in a document the execution is repeated up to the first `BreakPoint` in the document.

Syntax: `\BreakPoint([level,name]);`

The arguments `level` and `name` are optional. Using `level` we can determine the number of higher level environments which will be monitored. By default, the argument is `0`, i.e., only the variables of the active environment are displayed.

```
\variable X = 0;
\code{
      for (variable i = 1; i < 2; ++i)
      { BreakPoint(2);}
}
```

In this example, level 0 only contains the counting variable `i`. Level 1 applies to the `code` block that does not contain further variables. Specifying `level=2` also displays the value of the variable `X`.

If one uses several `BreakPoint` commands it proves to be useful to assign names to them. This makes it easier to identify the corresponding positions in the source code.

Example 6.12 Let us implement the Newton algorithm to determine a root of a given polynomial in SigMath. It is an iterative procedure with the following updating rule:

$$x_n = \begin{cases} \text{initialValue} & \text{if } n = 0 \\ x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})} & \text{if } n \in \mathbb{N} \end{cases}$$

While the iteration is running we want to monitor the values of x_n and $f(x_n)$. The polynomial is given by

$$f(x) = x^3 - 6x^2 - 4x + 24$$

and we choose `x = 3.5` as initial value.

For this we define a global variable `x` that contains one root of the polynomial at the end of the calculations. The functions `f(x)` and `fDerivative(x)` calculate the function value and the value of the derivative at the point `x`. The iteration is implemented in the function `Newton(x)` which is called from the `document` block and contains a variable to store the returning value.

Finally, using `BreakPoint` we can monitor x_n. The values of `x` and `FunctionValue` are displayed after every call of the function `Newton(x)`.

Code: (Newton.txt)

```
\variable x = 3.5;

\function f(x) = { return x^3 - 6*x^2 - 4*x + 24; }

\function fDerivative(x) = { return 3*x^2 - 12*x - 4; }

\function Newton(x) = { return x - f(x)/fDerivative(x); }

\begin{document}
 \variable FunctionValue = 0.0;
 \BreakPoint("Iteration");
 \x             = Newton(x);
 \FunctionValue = f(x);
\end{document}
```

The first interruption of execution is performed *after* the first call of `Newton(x)`. Thus, `x` was already updated and contains the value `1.27`.

During compilation `BreakPoint` has no effect. SigMath just gives a warning message. All information is saved to the file

`ProgDir\Shared\Temp\BreakPoint.txt`.

In order to compile and display the content of this file in SigMath one has to load

`ProgDir\Shared\GUI\FormatBreakPoint.txt`.

7 Computer graphics

This chapter should be seen as a starting point for OpenGL programming. We start with a short introduction to OpenGL and then consider the building blocks for every complex graphical object, the *primitives*. Primitives are, e.g., vertices, lines and polygons. Using this basic items we can generate three dimensional sceneries. The Section 7.4 introduces transformations and illustrates the way to create scenery. The usage of textures, i.e., bitmaps to decorate surfaces, is then treated in Section 7.7. Finally, we want to provide a small glimpse on the vast amount of methods to generate computer graphics provided by OpenGL.

7.1 What is OpenGL?

OpenGL stands for *Open Graphics Library* and was developed by Silicon Graphics. It provides a platform independent and programming language independent interface to use the graphics card. The library only contains graphics operations. In order to write code to generate computer graphics with OpenGL we have to make this interface accessible. We do not have to take care of this link since SigMath is doing this for us. All OpenGL commands can be identified by the prefix `gl.` Other SigMath commands, e.g., for word processing or programming do not belong to OpenGL. They can be used as a framework to generate graphics.

At the moment the library provides about 250 commands to display real-time animations of 3D sceneries. It does not contain commands to generate complex three-dimensional objects, though. All figures have to be built up from primitives. In order to draw a circle, e.g., one has to use a polygon. SigMath provides a number of commands that allow to display certain common figures (see Section 7.2.6). In addition, there is the *OpenGL Utility Library*, short `glu`, which allows to generate more complex graphical elements such as balls or cylinders (see Section 7.2.8). It is contained in the OpenGL package. Thus SigMath's graphics framework eases the task of generating graphics.

Let us describe the way OpenGL works. OpenGL is a *finite state machine*, i.e., the way a scene is presented is controlled by parameters of the current states. As examples may serve the material or position of an object or if the light is turned on or off. Most OpenGL commands change states. If an object is moved, not the whole scenery has to be redrawn. Only states are updated and the representation of the scene is recalculated.

7.2 Primitive

In computer graphics elementary graphical objects are called *primitives*. Elementary objects are vertices, lines or polygons. As already mentioned in the introduction, we can build complex

graphical objects from primitives. We start with vertices and learn how to generate primitives in SigMath. We list some predefined functions and `glu` commands to generate geometrical figures. We close with a elaborate example that generates a three-dimensional scenery.

7.2.1 Vertices

A *vertex* is an invisible point in a three-dimensional coordinate system. Here, "point" means the mathematical object, i.e., it has no size. An object in OpenGL is described by a set of vertices, i.e., the way it looks and the position of the object is determined by its vertices. They play a fundamental role in computing the scenery, the so-called *rendering* process. This computes light effects and color gradients depending on the vertices of an object.

Syntax:

```
\gl.Vertex(P.Vertex());
\gl.Vertex(x,y,z);
```

Coordinates of a vertex can be given on two ways. The first option uses a variable `P` of type `GLPoint` which has to be converted to an appropriate format using the `Vertex()` command. The second option uses the three coordinates `x`, `y`, and `z` with respect to SigMath's default coordinate system, directly. The origin `(0,0,0)` of the system is located in the lower left corner of the screen. The x-axis is pointing to the right, the y-axis to the top, and the z-axis points out of the monitor plane to the front.

Remark: In this section we always refer to this default coordinate system. For sceneries, we introduce further coordinate systems in Section 7.6.

7.2.2 OpenGL primitives

Let us turn to the primitives in OpenGL. As was already mentioned before, the shape of a geometrical object is defined by a list of vertices. In order to specify how OpenGL has to interpret this list we have to specify the type of primitive we want to describe.

Example 7.1 We give a list of 4 vertices. If we choose the object to be a line, the 4 vertices generate 2 independent lines. The vertices are left and right endpoints of the lines, respectively. If we choose the object to be a polygon instead, the same list of vertices results in drawing a quadragon.

There are zero-, one-, and two-dimensional primitives. We provide a list of all available types in Table 7.1. Although triangles and quadragons are polygons, they represent very special cases within the two-dimensional primitives. The reason is that such objects are used quite often. We give a more detailed description of these objects in the corresponding subsections.

Dimension	Object	Type of primitive
0	points	gl.POINTS
1	lines	gl.LINES,gl.LINE_STRIP, gl.LINE_LOOP
2	triangles	gl.TRIANGLES,gl.TRIANGLE_STRIP, gl.TRIANGLE_FAN
	quadragons	gl.QUADS,gl.QUAD_STRIP
	polygons	gl.POLYGON

Tbl. 7.1: Types of primitives

7.2.3 Implementation of primitives

In this section we give an insight in the way primitives are implemented in SigMath. Let us have a look at the following template in SigMath's online help.

Template (primitive)

```
\gl.PolygonMode(gl.FRONT_AND_BACK, gl.FILL);
\gl.Begin(gl.TRIANGLES);
  \gl.Color(1, 0, 0);
  \gl.Vertex(200, 200, 0);
  \gl.Color(0, 1, 0);
  \gl.Vertex(200, 400, 0);
  \gl.Color(0, 0, 1);
  \gl.Vertex(400, 400, 0);
\gl.End();
```

With `PolygonMode` the mode of the polygon is specified.

Syntax: `\gl.PolygonMode(side, mode);`

The argument `side` may take one of the following values:

`gl.FRONT`, `gl.BACK`, `gl.FRONT_AND_BACK`.

This argument determines which side of the polygon is drawn and thus depends on the value of `mode`. Mode can take the following values:

`gl.POINT`, `gl.LINE`, `gl.FILL`.

On `gl.POINT` only the vertices are drawn, `gl.LINE` adds connecting lines, and the mode `gl.FILL` draws lines and fills the whole area. We come back to the discussion of front and backside of polygons (and hence to the meaning of `side` in Section 7.2.5. The current settings of the mode of the polygon are valid till the next `PolygonMode` occurs. Thus, one does not have to specify it for every primitive, separately. The primitive environment is enclosed by `begin(primitiveType)` and `End()`. The argument `primitiveType` may be one of the values listed in Table 7.1.

Within this environment we specify a list of vertices. Depending on the type of the primitive one environment may be used to define more than one object of the same type. In case of `gl.TRIANGLES`, 6 vertices result in two triangles. Extending the list in type `gl.POLYGON` results in adding vertices to the polygon. Thus, only one polygon can be specified within one environment.

As one can see in the template also the color of the primitive is declared. All following objects are drawn using this color until the next `Color` command changes the color. Thus, every vertex might have another color. On default settings, different colors result in color gradients between the vertices which are computed during rendering process. Interesting effects already occur in very simple situations. OpenGL provides means to modify the computation of color gradients. We will not treat further methods and restrict our considerations to the default behavior, only.

7.2.4 Points and lines

`gl.POINTS` are the only representers of zero-dimensional primitives. On contrast to vertices, points have a diameter of one pixel.

Example 7.2

```
\gl.Begin(gl.POINTS);
 \gl.Vertex(100, 100, 0);
 \gl.Vertex(300, 100, 0);
 \gl.Vertex(150, 300, 0);
\gl.End();
```

One vertex always generates one point. Thus, within one environment arbitrary many points may be defined.

One-dimensional primitives are lines. A line is determined by its left and right endpoint. We write them as pair (left,right) and identify the points with their position in the list of vertices. For example, `(1,2)` defines a line that connects the first vertex to the second. OpenGL distinguishes three types of primitives for lines:

`gl.LINES` is the simplest version: any two consecutive vertices are connected as left and right end point of a line, respectively.

Example 7.3

```
\gl.Begin(gl.LINES);
 \gl.Vertex(100, 100, 0);
 \gl.Vertex(300, 100, 0);
 \gl.Vertex(300, 100, 0);
 \gl.Vertex(200, 300, 0);
 \gl.Vertex(200, 300, 0);
 \gl.Vertex(100, 100, 0);
```

```
\gl.End();
```

Here, three independent lines are given. Those are (1,2), (3,4), and (5,6). The lines form a triangle. One environment may contain arbitrary many lines.

`gl.LINE_STRIP`: On determining n vertices the second up to $(n-1)$th vertices are interpreted as left and right endpoints, respectively. This results in one connected line consisting of $n-1$ pieces. The first and last vertex give starting and endpoint of the polyline.

Example 7.4

```
\gl.Begin(gl.LINE_STRIP);
 \gl.Vertex(100, 100, 0);
 \gl.Vertex(300, 100, 0);
 \gl.Vertex(200, 300, 0);
 \gl.Vertex(100, 100, 0);
\gl.End();
```

Those 4 vertices result in the lines (1,2), (2,3), and (3,4). The result matches the object in Example 7.3 but requires only 4 vertices. One environment may contain only one polyline.

`gl.LINE_LOOP`: In contrast to the polyline `gl.LINE_STRIP` the first and last endpoint are also connected by a straight line, which results in a closed polyline of n vertices, connected by n lines.

Example 7.5

```
\gl.Begin(gl.LINE_LOOP);
 \gl.Vertex(100, 100, 0);
 \gl.Vertex(300, 100, 0);
 \gl.Vertex(200, 300, 0);
\gl.End();
```

This generates the same result as in Example 7.5 saving another vertex. Now the lines (1,2), (2,3), and (3,1) generate the triangle. The last vertex is the left endpoint of the last line and the first vertex its right endpoint. Thus, every vertex is left and right endpoint of a line. Multiple definitions in one environment are not allowed.

7.2.5 Surfaces

Surfaces are two-dimensional objects and thus the most complex primitives in OpenGL. Triangles and quadragons are the most frequent polygons and thus have a special role under all polygons.

Every surface has a front and a backside. This distinction has effect on computing sceneries and thus requires special care. Thus, we start with the determination of front and backside

before treating the available types.

Front and backside of polygons

Typically, the front side of a polygon is given as the side its normal vector is pointing at. Thus, we have to determine the positive direction of the normal of a surface.

Basically, there are two ways to determine this direction. The first variant is called *implicit definition of the normal vector*, which is important to draw the polygon. As already noted the arguments of `PolygonMode` allow to determine which side of the polygon has to be drawn. The determination of the sides of a polygon depends on the order of the vertices. The front is determined by using the right hand rule. We imitate the direction of rotation of the vertices using the fingers of our right hand . The direction the thumb is pointing at then gives the positive direction of our normal vector.

The second variant uses *explicit definition of the normal vector* in the definition of the primitive. Using light sources, this information is mandatory for the computation of the scenery and cannot be performed implicitly. Since we do not go into the details here, we just state the syntax to define the normal vector.

Syntax:

```
\gl.Begin(areatype);
 \gl.Normal(x,y,z);
 \gl.Vertex(x,y,z);
     ⋮
\gl.End();
```

We will illustrate the meaning of front and backside in connection with `PolygonMode` in Section 7.2.9.

Triangles

A triangle is uniquely determined by three points and is given as a triple (1,2,3). Since three points in three-dimensional space always lie in one plane no additional considerations are necessary. As with lines, SigMath distinguishes three types of triangles:

`gl.TRIANGLES` is the simplest type: Three consecutive vertices are connected to form a triangle.

Example 7.6

```
\gl.Begin(gl.TRIANGLES);
 \gl.Vertex(100, 100, 0);
 \gl.Vertex(300, 100, 0);
 \gl.Vertex(200, 300, 0);
```

```
 \gl.Vertex(300, 100, 0);
 \gl.Vertex(400, 300, 0);
 \gl.Vertex(200, 300, 0);
\gl.End();
```

In this example, two separate triangles (1,2,3) and (4,5,6) are drawn. The two triangles have two vertices in common, which have to be specified in `gl.TRIANGLES` for each triangle, separately. On the other hand, this allows to define multiple triangles in one environment.

`gl.TRIANGLE_STRIP`: As with `gl.LINE_STRIP` consecutive triangles glue together. The last two vertices of the previous triangle are the first two vertices of the following one, but in reversed order.

Example 7.7

```
\gl.Begin(gl.TRIANGLE_STRIP);
 \gl.Vertex(100, 100, 0);
 \gl.Vertex(300, 100, 0);
 \gl.Vertex(200, 300, 0);
 \gl.Vertex(400, 300, 0);
\gl.End();
```

This definition results in two triangles (1,2,3) and (3,2,4), yielding the same object as in Example 7.6. Every third vertex results in a further triangle. Thus, only several *connected* triangles can be defined in one environment.

`gl.TRIANGLE_FAN`: The first vertex is common vertex to all triangles in the primitive. It is always the first vertex in any triangle. The second vertex in every triangle is given as the last vertex of the previous triangle. The third vertex of one triangle is then the following vertex, i.e., for the second triangle the fourth vertex in the list and for the third triangle the fifth in the list, and so on. Only the first triangle is given by the first three vertices in the list in the mentioned order. This type may be used to draw a fan (therefore the name) or to draw a wheel with spokes.

Example 7.8

```
\gl.Begin(gl.TRIANGLE_FAN);
 \gl.Vertex(200, 300, 0);
 \gl.Vertex(100, 100, 0);
 \gl.Vertex(300, 100, 0);
 \gl.Vertex(400, 300, 0);
 \gl.Vertex(300, 500, 0);
 \gl.Vertex(100, 500, 0);
 \gl.Vertex(  0, 300, 0);
 \gl.Vertex(100, 100, 0);
\gl.End();
```

This example results in a ”wheel” consisting of 6 triangles: (1,2,3), (1,3,4), (1,4,5), (1,5,6), (1,6,7), and (1,7,8). As before, only one object can be defined in one environment.

Quadragons – Quads

A quadragon is defined via its 4 vertices. Now, 4 points in three-dimensional space do not have to lie in the same plane, but in a quadragon they have to. Violating this condition may result in incorrect display of the quadragon. Since the condition is not checked automatically, we have to take care that our vertices meet the requirement ourselves.

There are two types of primitives for quadragons:

`gl.QUADS`: This is the simple type. Similar to `gl.TRIANGLES` every four consecutive vertices are connected to form a quadragon.

Example 7.9

```
\gl.Begin(gl.QUADS);
 \gl.Vertex(100, 100, 0);
 \gl.Vertex(200, 100, 0);
 \gl.Vertex(200, 200, 0);
 \gl.Vertex(100, 200, 0);
 \gl.Vertex(100, 200, 0);
 \gl.Vertex(200, 200, 0);
 \gl.Vertex(200, 300, 0);
 \gl.Vertex(100, 300, 0);
\gl.End();
```

This example produced two quadragons (1,2,3,4) and (5,6,7,8) where the vertices 3 and 6, and 4 and 5 coincide. One environment can be used to define more than one object.

`gl.QUAD_STRIP`: In this type, two consecutive quadragons glue together. The first four vertices build the first quadragon. Every following quadragon uses the last two vertices of the preceding one in reverse order and the following two vertices of the list.

Example 7.10

```
\gl.Begin(gl.QUAD_STRIP);
  \gl.Vertex(100, 100, 0);
  \gl.Vertex(200, 100, 0);
  \gl.Vertex(200, 200, 0);
  \gl.Vertex(100, 200, 0);
  \gl.Vertex(200, 300, 0);
  \gl.Vertex(100, 300, 0);
```

```
\gl.End();
```

This definition generates two quadragons (1,2,3,4) and (4,3,5,6) resulting in the same geometrical object as in Example 7.9. We save two vertex definitions. This type allows to define one object per environment, only.

Polygon

A polygon is a surface having arbitrary many vertices. As before we have to take care of the fact that all points lie in one plane. This class contains only one type of primitive.

`gl.POLYGON`: Consecutive vertices are connected to form one polygon.

Example 7.11

```
\variable n = 10;
\variable D = 2.0*Pi/n;
\variable End = 2.0*Pi - D/2;
\gl.Begin(gl.POLYGON);
 for (variable F = 0.0; F < End ; F += D)
   gl.Vertex(r*cos(F), r*sin(F), 0);
\gl.End();
```

This example illustrates how to approximate a circle using a polygon. The quality of the approximation is controlled by the number of vertices (here $n = 10$). As before, only one object may be defined within one environment.

7.2.6 Additional functions

SigMath provides some additional functions, like, e.g., to draw circles or parts of the circle, to ease the tasks of generating scenery. This functions are contained in the directory

`ProgDir + "\Shared\OpenGL"`.

The functions can be accessed by using the `Execute` command as described on p. 20. Table 7.2 (on p. 150) contains a list of all currently available SigMath documents together with the functions they contain.

7.2.7 Line properties

The style of lines can be adjusted by modifying line properties. We start with the line width.

Syntax: `\gl.LineWidth(width);`

The argument `width` is of data type `Real` and is interpreted as pixels. By default, line width is set to 1. The admissible range for the line width can be retrieved with the help of the function

`gl.Get`. The same function also may be used to detect the smallest difference that is realizable, the so called *granularity*. Both values/ranges depend on the installed OpenGL implementation on the system.

Syntax:
```
\gl.Get(LINE_WIDTH_RANGE);
\gl.Get(LINE_WIDTH_GRANULARITY);
```

The first request on the range returns a one-dimensional `RealArray` with two elements: the first field gives the minimal and the second field the maximal line width. The granularity is also given as a `Real` constant.

Line smoothing can be activated or deactivated using the following commands:

Syntax:
```
\gl.Enable(gl.LINE_SMOOTH);
\gl.Disable(gl.LINE_SMOOTH);
```

If line smoothing is activated, the line width is automatically rounded up to the next integer.

Using `gl.LineStipple` allows to define line patterns.

Syntax: `\gl.LineStipple(factor, pattern);`

The argument `factor` is of type `Integer` which determines the way the pattern is stretched. The argument `pattern` requires a `String` of length 16 which consists of ”0” and ”1”, representing ”empty” and ”drawn” pixels, respectively. The usage of line patterns has to be activated or deactivated using the following two commands.

Syntax:
```
\gl.Enable(gl.LINE_STIPPLE);
\gl.Disable(gl.LINE_STIPPLE);
```

Example 7.12 The following lines of code generate three lines. The first line is dashed, the second appears as dotted line, and the third is a solid line. See Figure 7.1.

```
\gl.LineStipple(1,"1111000011110000");
\gl.Enable(gl.LINE_STIPPLE);
\gl.Begin(gl.LINES);
 \gl.Vertex(100,100,0);
 \gl.Vertex(250,250,0);
\gl.End();
\gl.Disable(gl.LINE_STIPPLE);
```

```
\gl.LineStipple(1,"1010101010101010");
\gl.Begin(gl.LINES);
 \gl.Vertex(100,200,0);
 \gl.Vertex(250,350,0);
\gl.End();

\gl.Disable(gl.LINE_STIPPLE);
\gl.Begin(gl.LINES);
 \gl.Vertex(100,300,0);
 \gl.Vertex(250,450,0);
\gl.End();
```

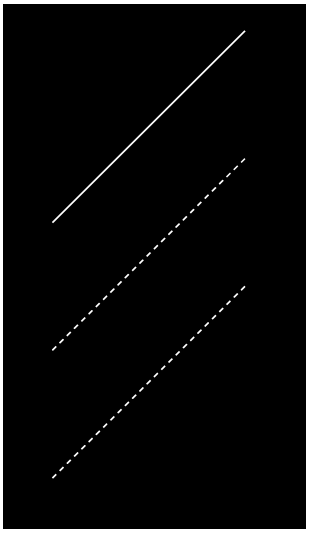


Fig. 7.1: Lines with different line patterns.

In a similar way we can generate patterns to fill up polygons.

Syntax: `\gl.PolygonStipple(pattern);`

Here `pattern` is a bitmap of type `gl.LUMINANCE`, `gl.DEPTH_COMPONENT` or `gl.ALPHA`. All three types consist of bitmaps of 128 pixels and every pixel is given by one byte (8 bit). Thus we have 128×8 bits to generate a 32×32 pixel pattern. As for the lines, the `pattern` consists of zeros and ones which are interpreted in the same way as before.

Example 7.13 INSERT EXAMPLE FOR POLYGON WITH PATTERN

7.2.8 Quadrics

In the beginning we learned how to draw points, lines, and surfaces. From this objects we can build more complex geometrical objects, e.g., *quadrics*. To ease the task of generating quadrics, the `glu` library provides further functions to draw spheres, cylinders or pointed disks. These functions generate the geometrical object from primitives based on the values of the arguments.

Remark: The way to scale, move or rotate quadrics will be introduced in Section 7.4.

Spheres

The function `glu.Sphere` generates a sphere from quadragons and triangles.

Syntax: `\glu.Sphere(radius, longitudinalScaling, latitudinalScaling);`

The model for the sphere is the globe. The poles are automatically generated and the connecting line runs along the z-axis of the global coordinate system. This way, longitude and latitude is defined. At the poles the surface of the sphere is constructed from triangles, in between from quadragons. In summary, the sphere consists of `longitudinalScaling`×`latitudinalScaling` pieces.

Cylinders

The function `glu.Cylinder` allows to draw cones.

Syntax:

```
\glu.Cylinder(radiusBottom, radiusTop, height,
              horizontalScaling, verticalScaling);
```

The cone rests on the xy-plane of the global coordinate system and the top is pointing in the direction of negative z-axis. The `horizontalScaling` gives the resolution around the z-axis, the `verticalScaling` the resolution along the z-axis. One can draw cylinders and pointed cones, depending on `radiusBottom=radiusTop` and `radiusTop=0`, respectively.

Disks and disk segments

A disk is a round surface having a hole in the middle, like a CD. They can be drawn using `glu.Disk`.

Syntax:

```
\glu.Disk(innerRadius, outerRadius,
          horizontalScaling, radialScaling);
```

The arguments `innerRadius` and `outerRadius` determine the radii of the hole and the disk, respectively. The center of the disk is located at the origin of the global coordinate system

and this disk lies in the xy-plane. The `horizontalScaling` equals the number of slices, the `radialScaling` gives the number of rings, the disk consists of.

The function `glu.PartialDisk` is a modification of this command.

Syntax:
```
\glu.PartialDisk(innerRadius, outerRadius, horizontalScaling,
                 radialScaling, initialAngle, cutSize);
```

This function allows to draw segments of a disk. We need two additional arguments in comparison to `glu.Disk`: The initial angle of the segment and its size. Both are given in degrees and measured in clockwise direction. One can imagine that we cut a slice out of the pie.

Properties of quadrics

With the help of the command `glu.QuadricDrawStyle` we can specify certain general properties to draw quadrics.

Syntax: `\glu.QuadricDrawStyle(property);`

The possible values of `property` and its meaning are the following:

`glu.FILL`	surfaces are filled
`glu.LINE`	boundaries of surfaces are drawn
`glu.POINT`	only points are drawn
`glu.SILHOUETTE`	only edges are drawn

Since quadrics are built up from primitives, all specifications in `gl.PolygonMode` take effect on the way they are displayed (see Section 7.2.3). The following applies to quadrics: if both properties are set, the one that draws less is applied.

Example
```
\gl.PolygonMode(gl.FRONT_AND_BACK, gl.LINE);
\glu.QuadricDrawStyle(glu.FILL);
```

In this example, only lines are drawn.

We already mentioned the normal vector of a surface in connection with front and backside of polygons. Every quadric consist of surfaces, and thus, we can determine its normal vector. We do not determine the normal vector for each surface separately, but by defining the properties of the normal vector. The rest is performed by the following `glu` function.

Syntax: `\glu.QuadricOrientation(arg);`

This command sets the direction of the normal vector and thus specifies front and backside of the surfaces. The argument takes the value `glu.OUTSIDE` if the normal vectors should point to the outside. Thus, on `glu.OUTSIDE` the front sides of the surfaces all point outwards. Alternatively, on `glu.INSIDE` the normal vectors are pointed inwards. The default setting is `glu.OUTSIDE`.

Remark:

1. For disks `glu.OUTSIDE` results in a normal vector pointing in the direction of the positive z-axis.
2. The function `glu.QuadricNormals` is important for the later rendering process. We do not go into further details, here.

7.2.9 An extended example

INSERT SCENE WITH PRIMITIVES AND ILLUSTRATE THE MEANING OF FRONT AND BACKSIDE

7.3 Working with lists

With the help of lists one can group several OpenGL commands. Besides improving readability of code it allows to display scenery efficiently. In addition, once we defined a scenery in a list we can draw the scenery by simply calling the list and do not have to rewrite the commands.

7.3.1 Generating lists

To generate a display list one has to write the following:

Syntax:

```
\variable listName = gl.GenList(numList);
\gl.NewList(listName, listMode);
gl-commands
\gl.EndList();
```

The variable `listName` is of type `Integer` and `gl.GenList` assigns a uniquely determined identifier to it. The argument `numList` is also of type `Integer`. Simply writing `variable listName = 1` is also possible but the usage of `gl.GenList` guarantees that every identifier is unique and, thus, prevents lists from being overwritten. `gl.NewList` opens the definition of the list. The argument `listMode` can take the values `gl.COMPILE` and `gl.COMPILE_AND_EXECUTE`. This results in either just compiling the list of compiling the list and executing the result, directly.

7.3.2 Calling lists

To call a list one uses the following commands:

Syntax:
```
\gl.CallList(listName);
\gl.CallLists(listNameArray);
```

One can either give one `Integer listName` or a `IntegerArray listNameArray` to call one or multiple lists, respectively.

To prevent runtime errors, one can check the existence of a list with `gl.IsList`.

Syntax: `\gl.IsList(listName);`

This function returns `true` if the list exists and `false`, otherwise.

Lists may contain lists. This allows to generate complex sceneries from sub-sceneries which are already given as lists.

Example 7.14 HERE AN EXAMPLE FOR A LIST

7.4 Transformations

Transformations include scaling, translation, and rotation of points. A transformation changes the so-called active OpenGL matrix which is applied to all following objects before they are drawn. Application means multiplying the matrix with the points. We will explain the behavior of these 4×4-matrices in Section 7.5 and give some examples for transformations of objects, there.

7.4.1 Scaling

Scaling an object results in stretching or squeezing an object.

Syntax: `\gl.Scale(sx, sy, sz);`

`sx`, `sy`, and `sz` are of type `Real` and positive. If one of the values is between 0 and 1 the object is squeezed, if the value is bigger than one the object is stretched in this direction. Scaling is implemented as the following matrix multiplication.

$$\begin{pmatrix} \texttt{sx} & 0 & 0 & 0 \\ 0 & \texttt{sy} & 0 & 0 \\ 0 & 0 & \texttt{sz} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \texttt{sx} \times x \\ \texttt{sy} \times y \\ \texttt{sz} \times z \\ 1 \end{pmatrix}$$

7.4.2 Translation

On translations we move an object along a straight line which is given by a vector. One can imagine that the vector is attached to every point and then the points are moved along the vector.

Syntax: `\gl.Translate(dx, dy, dz);`

To move a point, we just have to add the vector `(dx,dy,dz)`. The corresponding matrix multiplication reads as follows:

$$\begin{pmatrix} 1 & 0 & 0 & \texttt{dx} \\ 0 & 1 & 0 & \texttt{dy} \\ 0 & 0 & 1 & \texttt{dz} \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + \texttt{dx} \\ y + \texttt{dy} \\ z + \texttt{dz} \\ 1 \end{pmatrix}$$

7.4.3 Rotation

A rotation is given by a rotation axis and a rotation angle. The rotation angle is given is degrees, the rotation axis is specified by a vector where the direction determines the direction of rotation. It is given implicitly by the right hand rule (see p. 126).

Syntax: `\gl.Rotate(phi,x,y,z);`

The rotation matrix depends on the rotation angle and the rotation axis. As an example we give the matrix multiplication for a rotation around the positive z-axis:

$$\begin{pmatrix} \cos(\texttt{phi}) & -\sin(\texttt{phi}) & 0 & 0 \\ \sin(\texttt{phi}) & \cos(\texttt{phi}) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\texttt{phi}) \cdot x - \sin(\texttt{phi}) \cdot y \\ \sin(\texttt{phi}) \cdot x + \cos(\texttt{phi}) \cdot y \\ z \\ 1 \end{pmatrix}$$

This multiplication corresponds to `gl.Rotate(phi,0,0,1)`.

7.5 OpenGL matrices

There are three different types of OpenGL matrices which can be activated using one of the following constants:

`\gl.MODELVIEW`	positioning of the camera
`\gl.PROJECTION`	adjustment of the camera
`\gl.TEXTURE`	texture processing

The first two types will be introduced in connection with the OpenGL viewing pipeline in Section 7.6. The matrix type `gl.TEXTURE` is treated in Section 7.7. This section only treats the general functionality and introduces matrix stacks.

7.5.1 Functionality of OpenGL matrices

Matrices are used in connection with transformations. Without going into the details of modelview matrices we will use them in examples to transform objects. The following example illustrates the transformation of a quadric.

Example 7.15

```
\gl.MatrixMode(gl.MODELVIEW);
\gl.LoadIdentity();
\glu.QuadricDrawStyle(glu.LINE);
\glu.Sphere(100,10,10);
\gl.Translate(200,200,-200);
\glu.Sphere(100,10,10);
\gl.Translate(300,300,0);
\gl.Scale(2,2,2);
\gl.Rotate(45,1,0,0);
\glu.Sphere(100,10,10);
```

The first command activates the modelview matrix and sets its current value to the identity matrix. Then we draw a sphere with center lying in the origin. The next sphere is generated using the same arguments as the first one. Issuing the `Translate` beforehand moves the sphere such that its center is located at `(200,200,-200)`. Then we perform another translation, scale by the factor 2 and rotate by 45 degrees around the x-axis. The center of the last sphere has coordinates `(500,500,-200)`, its radius is 200 and the poles are connected along the bisectrix of the yz-plane.

All these operations are implemented as matrix multiplications from the left by the corresponding transformation matrix (see Section 7.4). Before an object is drawn, its vertices are multiplied by the active OpenGL matrix. In the example, the coordinates of the third sphere are multiplied by the result of the following matrix multiplication:

$$\text{identity} \times \text{translation} \times \text{translation} \times \text{scaling} \times \text{rotation}$$

7.5.2 Matrix stack

With `gl.MatrixMode` we actually do not refer to *one* matrix, but to a whole stack of matrices. Its depth is constant in Example 7.15. Let us consider the general case, here.

In stacks, we can only access the topmost entry. In case of the OpenGL matrix stack this is the current active OpenGL matrix. We can use the following commands to modify the stack:

Syntax:

```
\gl.PushMatrix();
\gl.PopMatrix();
```

With `gl.PushMatrix` we can add a new matrix on top of the stack, while `gl.PopMatrix` deletes the topmost matrix. The advantage of using the stack structure is that we do not have to reset the transformation matrix to identity for a new transformation. If we need the current active matrix later on, we just push a new matrix to the stack. This allows to perform a transformation, which is independent of the former ones, without having to discard the previous information.

Example 7.16

```
\glu.QuadricDrawStyle(glu.LINE);
\gl.MatrixMode(gl.MODELVIEW);
\gl.LoadIdentity();
\gl.Translate(200,200,-200);
\glu.Sphere(100,10,10);
\gl.PushMatrix();
\gl.LoadIdentity();
\gl.Translate(600,600,0);
\glu.Sphere(100,10,10);
\gl.PopMatrix();
\gl.Scale(2,2,2);
\glu.Sphere(100,10,10);
```

In this example, the second sphere has its own transformation matrix without having to delete the first one. The latter is turned active by deleting the additional matrix from the stack on `gl.PopMatrix`. This results in the fact that the centers of the first and last sphere coincide.

7.6 Scenery

This section is devoted to the generation of scenery, the so called *OpenGL Viewing Pipeline*. Composing a scenery is performed in several steps that will be explained in detail. First, we give a schematic view on the whole process.

First, the *object coordinates* are multiplied by the modelview matrix (`gl.MODELVIEW`). Object coordinates are the coordinates which were specified in the definition of the primitives used for this object. They are referring to the usual global coordinate system having its origin in the bottom left corner of the screen. This multiplication can be interpreted as viewing the object through the lense of a camera. Thus, the modelview matrix is used to determine the position of the camera. Further options to modify the position of the camera will be treated in Section 7.6.1.

The result of the multiplication of the object coordinates by the modelview matrix are the so-called *eye coordinates*. The camera is placed at the origin of this coordinate system. Using a projection matrix (`gl.PROJECTION`) we generate the cuboid we want to look at. It is then given in the *clip coordinate system*. The transformation matrix between the eye and the clip coordinates is just given by the projection matrix. More information on the technical details is

given in Section 7.6.2.

On basis of the chosen perspective (see Section 7.6.2) the normalized device coordinates are computed. Here sizes – and thus coordinates – are determined according to the distance of the object to the observer.

Finally, the *window coordinates* are the result of the viewpoint transformation. This determines the area of the screen which contains the scene we want to display. The scene is automatically scaled to the size of the viewport. Section 7.6.3 will give more details on this last step.

7.6.1 Positioning of the camera

Basically, the position of the camera is fixed at the origin pointing in the direction of the negative z-axis. Nevertheless, every transformation of objects can also be regarded as a transformation of the camera and, thus, we speak of positioning the camera. This positioning is performed using the modelview matrix. On the level of implementation we have two choices to perform the positioning:

1. We move all objects using transformations in front of the camera, as in Example 7.15. Since all transformations have to be given explicitely, we call this method *explicit*.
2. The `glu` library contains a command to position the camera: `glu.LookAt`. This can be used to move or rotate the camera. Since camera position is fixed, we implicitly move and rotate all objects. This command involves several matrix multiplications.

Syntax:

```
\glu.LookAt(eye.x, eye.y, eye.z, vc.x, vc.y, vc.z,
            vup.x, vup.y, vup.z);
```

The eye coordinates determine the position of the camera. `vc` stands for viewcenter and defines the center of the scene. The direction of view is pointing from the camera position to the center of the scene. The view-up vector `vup` determines which side is up. A view up vector pointing in the direction of the negative x-axis, e.g., results in rotating the scene by 90 degrees around the positive z-axis. All arguments are of type `Real`.

7.6.2 Projection

All transformations in this subsection are performed using a projection matrix. It is used to limit our considerations to a cuboid in three-dimensional space. This is done by determining the so called clip planes, i.e., the six faces of the cuboid. All objects lying outside this cuboid are neglected for the generation of the scenery. Besides the clip planes we have to specify the type of projection. There are two variants available.

Parallel projection: the camera focuses on an infinitely distant point. That means, the distance of an object to the observer does not affect the size of the object. Distance is only used to detect if one object is blocked from sight by another object in the front. This projection results in a two-dimensional view on the scene.

Syntax: `\gl.Ortho(left, right, bottom, top, near, far);`

The arguments are used to define the six clip planes: Starting from the previously fixed camera position we specify a minimal and maximal distance to the camera. This generates a rectangle, having the camera in its center. Finally, `near` and `far` specify near and far distance, which turns the rectangle into the cuboid. The orientation of the cuboid in space is given by the direction of view.
For the distances the following rules of sign apply: distances to the right, to the top or to the front (in direction of view) must be positive, distances to the left, to the bottom and to the back must be negative. The arguments `near` and `far` may take positive or negative values of type `Real`.

INSERT EXAMPLE OF ORTHOGONAL PROJECTION
The default values for all SigMath documents are given by

```
\gl.Ortho(0, W-1, 0, H-1, 1000, -1000);
```

where `W` and `H` denote the window width and window height given in pixels.

Perspective projection: A more realistic view. On a perspective projection the size of objects decreases on increasing distance to the camera. The generated sceneries are more realistic.

Syntax:

```
\gl.Frustum(left, right, bottom, top, near, far);
\glu.Perpective(visionY, ratio, near, far);
```

Determination of the visible area is performed using `gl.Frustum`. The arguments have the same meaning as in the parallel projection. The function `glu.Perspective` produces the same result as `gl.Frustum` but uses different arguments. `visionY` gives the angle of the top clip plane to the y-axis and `ratio` determines the ratio of width to height. `near` and `far` determine the front and back clip plane. In contrast to the previous usage, the values must be positive.

INSERT PERSPECTIVE PROJECTION

Based on the type of projection and the determination of the visible cuboid, OpenGL computes the clip coordinates. Those are used in the last step to determine the window coordinates which are actually used to display the scenery on the screen.

7.6.3 Viewport

Basis for the viewport transformation are the normalized device coordinates. In this step the scenery is adapted to technical limitations of the system: color gradients have to be reduced to a finite color palette, straight lines have to be displayed using pixels. The whole projection step is performed by OpenGL automatically. The viewport command defines a rectangular area in the window that contains the scenery.

Syntax: `\gl.Viewport(x, y, width, height);`

The coordinates `x` and `y` mark the bottom left corner of the display area, `width` and `height` specify its size. The ratio of width and height should match the given ratio in the projection commands in order to prevent distortion. The default values for all SigMath documents are given by the following command:

```
\gl.Viewport(0, 0, W, H);
```

where `W` and `H` denote the width and height if the window in pixels. The resulting coordinates give the window coordinates.

7.7 Textures

7.7.1 Basic principles

Images which are mapped to the surface of an object are called *textures*. The mapping process is called *texture mapping* and the whole process is also called *texturing*. Texturing can be used to model surfaces of objects by using image information. This technique allows to add more detail to a scene without having to change its geometrical complexity.

Example 7.17 Imagine we want to display a paved plaza. One attempt would be to generate every single stone in the pavement by primitives. To generate a more realistic scenery, we have to add structure to the surface of every stone. Modeling may get arbitrarily involved making the task to generate a realistic image impractical. Alternatively, we can model the whole paved area as one primitive and map an image of a pavement to the surface: we use textures.

This section will introduce the available types of textures and illustrate how to load and map textures. We focus on a hands-on introduction to textures and will not go into the theoretical details.

7.7.2 Types of textures

Textures can be classified based on their dimension. There are one-, two-, and three-dimensional textures. Here, we focus on two-dimensional textures. Besides *discrete textures*, where images are used as textures, there also exist *procedural textures*. In this case, the structure is generated

from a mathematical algorithm. Discrete textures are easy to use. A digital camera and image processing software allows to generate as much images as we want. Memory consumption is larger than for procedural textures, though. On the other hand, generating procedural textures may get very complicated. Basically, one rather uses procedural textures in regular structures and discrete textures for irregular, detailed structures.

7.7.3 Loading textures

In order to use a texture, we have to load it to video memory. If the available memory is small, data has to be stored in main memory and results in slowing down graphical display.

Discrete and procedural textures can be implemented in SigMath in a very similar way. We explain the procedures for discrete textures in detail and restrict to pointing out the difference in the case of procedural textures.

Discrete textures

In the text editor you can find several templates to load textures. We will have a look at two templates `TextureLoad` and `LoadTexture`, and illustrate the usage of `glu.Build2DMipMaps` and `gl.TexImage2D`.

Template

```
\function TextureLoad(filename) =
{
 variable TextureNr = gl.GenTextures();
 gl.BindTexture(gl.TEXTURE_2D, TextureNr);
 gl.TexParameter(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
 gl.TexParameter(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
 glu.Build2DMipmaps(filename);
 gl.BindTexture(gl.TEXTURE_2D, 0);
 return TextureNr;
}
\variable roofTile = TextureLoad(path + filename);
```

The argument `filename` is of data type `String` and contains the (full-qualified) path, filename and filename extension of the image which is used as texture. First we define a variable `TextureNr` in order to reference to the texture later on. The function `gl.GenTextures` guarantees that every texture has a uniquely determined identifier (of type `Integer`), also called *ID of the texture*. We could also assign an identifier manually. But then we have to check uniqueness by ourselves. `gl.BindTexture` binds the specified texture to the target surface.

Syntax: `\gl.BindTexture(target, texture);`

Besides connecting texture and surface, we also specify that our texture is two-dimensional. As soon as a texture is bound to an object, every manipulation of the object takes effect on the

texture, too. In this way, the target acts as a representer of the texture. In this introduction `target` is always specified by `gl.TEXTURE_2D`. The properties of the texture can be modified using `gl.TexParameter`.

Syntax: `\gl.TexParameter(target, parameter, value);`

By specifying `gl.TEXTURE_2D` as target all parameters of the current texture are modified. The current texture in the example is the texture with ID `TextureNr`. The different parameters, its possible values, and the effects of those values are summarized in the following table. These parameters can be used to control the way the texture mapping is performed. Points on the object are pixels, points in the texture are called *texels* (texture elements). The coordinates of the object are denoted by x and y, whereas the corresponding coordinates of the texture are denoted by s and t.

parameter	value
gl.TEXTURE_MAG_FILTER	gl.LINEAR gl.NEAREST
gl.TEXTURE_MIN_FILTER	gl.LINEAR gl.NEAREST gl.NEAREST_MIPMAP_NEAREST gl.LINEAR_MIPMAP_NEAREST gl.NEAREST_MIPMAP_LINEAR gl.LINEAR_MIPMAP_LINEAR
gl.TEXTURE_WRAP_S	gl.REPEAT gl.CLAMP
gl.TEXTURE_WRAP_T	gl.REPEAT gl.CLAMP

`gl.TEXTURE_MAG_FILTER`: If the texture is smaller than the surface, the texture must be enlarged. There are more pixels than texels making a one-to-one mapping impossible. This parameter controls the way the color of the pixel is calculated. On `gl.LINEAR` the color of the pixel is computed as a weighted average of four neighboring texels, whereas on `gl.NEAREST` we use the color of the nearest texel to the pixel.

`gl.TEXTURE_MIN_FILTER`: This parameter controls the way a pixel is colored if the texture is larger than the surface. The values `gl.LINEAR` and `gl.NEAREST` have the same effect as in the magnification filter. More information on the other possible values can be found in Section 7.7.4 on the mipmapping technique.

`gl.TEXTURE_WRAP_S`: This parameter controls if the texture shall be repeated horizontally. This way of repetition is also called *tiling*. Either we enable repetition (`gl.REPEAT`) or disable repetition (`gl.CLAMP`). Repetition is enabled by default.

`gl.TEXTURE_WRAP_T`: This parameter controls vertical tiling and behaves like the horizontal case.

After adjusting all texture parameters we have to bind the texture variable to the image in a file. Its file name is the argument to the function `TextureLoad`. Finally, `glu.Build2DMipmaps` loads the image to video memory.

Syntax: `\glu.Build2DMipmaps(filename);`

Width and height of the image (measured in pixels) have to be equal to a power of two. If this is not the case, the command automatically scales the image to the next larger admissible size.

Alternatively, we can specify a bitmap as argument in the function `glu.Build2DMipMaps`, instead.

Syntax: `\glu.Build2DMipmaps(image);`

Finally, `gl.BindTexture(gl.TEXTURE_2D, 0)` removes the bind of target and texture and `return TextureNr` returns the texture ID as return value of the function. The texture ID 0 is reserved and represents an initial texture. All targets are bound to this initial texture as long as we do not bind it to another texture ourselves.

In this way, we implemented a function to load textures. The return value of the function is the ID of the texture. Thus, `TextureLoad` is mainly used in the definition and initialization of variables. In our example, `roofTile` is then the identifier of the loaded texture.

As indicated, we introduce a second function, called `LoadTexture` that provides basically the same functionality as `TextureLoad`. In the implementation we use the OpenGL command `TexImage2D` instead of the `glu` command `glu.Build2DMipMaps`.

Syntax:

```
\gl.TexImage2D(image);
\gl.TexImage2D(filename);
```

There are two versions of this command. `image` is a variable of type `Bitmap` and `filename` is the full qualified name (path + filename + extension) of the file containing the image. This command does not rescale the given image and, thus, code has to be adjusted accordingly.

Template

```
\function LoadTexture(filename, w, h) = // w and h are redefined
{
 variable "bmp" Image = Bitmap();
 Image.LoadPower2(filename, w, h);
```

```
 variable TextureNr = gl.GenTextures();
 gl.BindTexture(gl.TEXTURE_2D, TextureNr);
 gl.TexParameter(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
 gl.TexParameter(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
 gl.TexImage2D(Bild);
 gl.BindTexture(gl.TEXTURE_2D, 0);
 return TextureNr;
}
\variable w = 0.0;
\variable h = 0.0;
\variable roofTile = LoadTexture(path + filename,w,h);
```

We note the two additional arguments in the function head. They have to be of type `Real` but are reset within the function body. Thus, their actual values are not used. The first command defines a `Bitmap` variable and `bmp.LoadPower2` loads the specified image to this variable.

Syntax: `\bmp.LoadPower2(filename, w, h);`

This command generates a `Bitmap`, where the side lengths equal the smallest possible power of 2 larger than the actual proportions of the loaded image. The image is placed to the bottom left corner of the `Bitmap` and the remaining pixels are colored with black (0). Finally, `w` and `h` are computed using the following formulas:

$$\texttt{w} = (\text{width of image})/(\text{width of Bitmap})$$
$$\texttt{h} = (\text{height of image})/(\text{height of bitmap}).$$

The values of `w` and `h` therefore are contained in the half-open interval $(0.5, 1]$. The reason for these assignments will be explained in Section 7.7.5. The remaining part of the function follows the corresponding lines in `TextureLoad`, where `glu.Bind2DMipmaps` is substituted by `gl.TexImage2D` which is called using the `Bitmap` variable containing the image, and not using the filename, directly.

Procedural textures

SigMath provides the data type `Bitmap` to store images. To generate a new `Bitmap` variable we use the following `Bitmap` command.

Syntax: `\variable image = Bitmap(type);`

By default, `type` has the value `gl.RGB`. This generates a `h`×`w`×3-array. `h` and `w` represent height and width of the image. The third dimension is used to store the RGB color coordinates. The function `LoadPower2` automatically fills the corresponding values from the file. Working with procedural textures, we fill the array manually by specifying for every single pixel its red, green, and blue color coordinates. We can access the single elements with index triples, as usual

for arrays. Mind that the first row of the array (index 0) is the bottom line of the image in display. Finally, `h` and `w` should be powers of 2 such that we can use `TexImage2D` without having to rescale the image beforehand.

Example 7.18 The following code generates a chess board.

Code: (Chess.txt)

```
\variable Image = Bitmap();
\code{
      Image.Resize(32,32);
      for(variable i=0; i<16; ++i)
        for(variable j=0; j<16; ++j)
          {Image(i,j,0) = 49; Image(i,j,1) = 49; Image(i,j,2) = 49;}
      for(variable i=16; i<32; ++i)
        for(variable j=16; j<32; ++j)
          {Image(i,j,0) = 49; Image(i,j,1) = 49; Image(i,j,2) = 49;}
      for(variable i=16; i<32; ++i)
        for(variable j=0; j<16; ++j)
          {Image(i,j,0) = 250; Image(i,j,1) = 250; Image(i,j,2) = 250;}
      for(variable i=0; i<16; ++i)
        for(variable j=16; j<32; ++j)
          {Image(i,j,0) = 250; Image(i,j,1) = 250; Image(i,j,2) = 250;}
}
```

The `Bitmap` variable `Image` can be used in the same way as the`Bitmap` variable we filled using the function `LoadTexture` with the help of `LoadPower2`.

Using text as textures

SigMath provides a very comfortable way to convert text into texture. This can be quite useful if we want to write some text onto the surface of an object.

Syntax: `\bmp.TextureFromString(text, fontBase, w, h);`

The function `TextureFromString` in the class `bmp` returns the ID of a texture that contains the `String text`. The argument `fontBase` is of type `Integer` and specifies which font should be used. Here 0 uses the current font. `w` and `h` determine width and height of the texture.

The texture matrix

We already encountered modelview and projection matrices in connection with transformations. Before objects are displayed, the object coordinates are multiplied by the modelview matrix. Similarly, the texture coordinates `(r,s,t,q)` are multiplied by the texture matrix before they are drawn. This allows to scale, translate, and rotate textures.

Remark: Since we only consider two-dimensional textures, the coordinates `r` and `q` can be neglected. Thus, we always assume them to be 0.

7.7.4 Mipmapping

INSERT

7.7.5 Texture mapping

Texture mapping denotes the process of mapping texture to an object. That means, that the coordinates `s` and `t` of the texture are mapped to coordinates `x` and `y` of the object. We have already seen that the values of texture parameters (see Section 7.7.3) have an effect on this procedure. In the following we will have a look at this mapping.

Texturing primitives

On the level of primitives, texture coordinates are mapped pointwise to the vertices of the primitives. Before we can do that, we have to enable texturing.

Syntax: `\gl.Enable(gl.TEXTURE_2D);`

Now, texture coordinates can be specified before the corresponding vertex in the primitive definition.

Syntax:
```
\gl.TexCoord(s,t);
\gl.Vertex(x,y);
```

Here, texture coordinates are in normalized form, i.e., `(0,0)` denotes the top left corner of the texture and `(1,1)` the bottom right corner. We close the texture mapping process by deactivating texturing.

Syntax: `\gl.Disable(gl.TEXTURE_2D);`

Example 7.19 EXAMPLE: MAP TEXTURE TO SQUARE

We already learned that the wrap parameters (see p. 143) control whether texture can be repeated. The number of repetitions is determined using the relative texture coordinates. If we assign `s` to `x` the texture is repeated `s` times until we reach `x`. The same applies to the vertical direction.

Example 7.20 MAP TEXTURE REPEATEDLY TO SQUARE

A texture coordinate smaller than 1 results in displaying only a part of the texture. If the texture coordinate is larger than 1 we repeat the texture.

Remark: This should explain why we set the values `w` and `h` in the function `TextureLoad` in the prescribed way (see p. 145).

Texturing quadrics

Since we do not define the vertices of a quadric explicitely, we cannot use the method described in the previous paragraph. Instead of defining the mapping pointwise, we use a function to generate texture coordinates for the texturing of quadrics.

Syntax: `\glu.QuadricTexture(condition);`

This `glu` function enables or disables the generation of texture coordinates for quadrics depending on the value of the Boolean argument `condition`. If it is disabled, all quadrics are automatically drawn using the current texture bound to it. The number of repetitions cannot be controlled with the help of the texture coordinates `s` and `t` directly. By default, `glu.QuadricTexture` generates the texture coordinates in such a way that the texture is mapped only once to the object. Since texture coordinates are multiplied by the texture matrix before displaying them, we can use this matrix to modify the coordinates.

Example 7.21 INSERT: SPHERE WITH TILED TEXTURE

On scaling, all s-values of the texture coordinates are multiplied by 4 and all t-values by 3. The tiling of the square in Example 7.21 may also be generated by using the texture matrix.

Reflection mapping

Reflection mapping allows to generate interesting mirror effects in a simple way. The effect can be interpreted in the following way: the texture is mapped to the inside of a sphere, that encloses the object. Then the texture is mapped towards the center of the sphere. This is done using spherical coordinates, but we do not give the details here.
On loading a texture we can specify a function that generates the texture coordinates.

Syntax: `\gl.TexGen(texCoordinate, genFunction, genParameter);`

`texCoordinate` determines, which texture coordinate is modified. The following values are possible: `gl.GL_S`, `gl.GL_T`, `gl.GL_R`, and `gl.GL_Q`. Since we treat only two-dimensional textures, we only use the first two. `genFunction` specifies the function that is used to generate the texture. To use reflection mapping, we have to use `GL_TEXTURE_GEN_MODE` and the parameter `genParameter=GL_SPHERE_MAP`.

Remark: There are several other complex mapping techniques, which can be used if we choose other values for the arguments in `gl.TexGen`.

If we want to generate texture coordinates we have to enable this procedure in addition to the general activation of texturing.

Syntax:

```
\gl.Enable(gl.TEXTURE_GEN_S);
\gl.Enable(gl.TEXTURE_GEN_T);
```

Finally, one should not forget to deactivate the texturing by using `gl.Disable(...)`.

Example 7.22 ANIMATED SPHERE WITH TREES (SIGMATH EXAMPLE)

The mirror effect is produced by using the background wallpaper as texture, too. This and many other examples can also be found in the Example section of SigMath.

Subdirectory	Filename	Functions
	KoordUtility.txt	SetPointKoordFromZyl() SetPointKoordFromSph() SetPointKoordFromKart()
Array	Brackets.txt	DrawBrackets() Brackets() BracketsPart()
	Rectangles.txt	DrawArrayRect()
	VLines.txt	VLineR()
Arrows	Arrow2D.txt	Arrow2D()
	Arrow2DDouble.txt	Arrow2DDouble()
	Arrow2DBezier.txt	Arrow2DBezier()
	Arrow2DBezierDouble.txt	Arrow2DBezierDouble()
	Arrow3D.txt	Arrow3D()
	Line.txt	Line()
	TubeClosed.txt	TubeClosed()
Bodies	Cube.txt	DrawCube()
	CubeTexture.txt	TexturedCube()
	CubeTheColor.txt	TheColorCube()
	Gear.txt Parallelepiped.txt	Gear() ParPiped()
	ParallelepipedTubes.txt	ParPipTubes()
Circle	Circle.txt	Circle2D()
	PartialCircleAA2D.txt	PartialCircleAA2D()
	PartialCircleArrow2D.txt	PartialCircleArrow2D()
	PartialCirclePA2D.txt	PartialCirclePA2D()
Ellipse	Ellipse.txt	Ellipse2D()
Figures	Ikosaeder.txt	Ikosaeder()
	RegularPolygon.txt	RegularPolygon()
	RegularPolygonRect.txt	RegularPolygonRect()
Lines	GridLines.txt	GridLines()
	LineBezier2D.txt	LineBezier2D()

Tbl. 7.2: provided functions

8 Using and Writing DLLs

Import of external libraries, so called *dynamic link libraries*, *DLL* for short, can increase the functionality of SigMath. We will give a short description of how to import existing DLLs or how to generate DLLs for this purpose.

8.1 Loading DLLs

To make functions or constants contained in an external DLL available within SigMath we use the command `syst.LoadLibrary`.

Syntax: `\syst.LoadLibrary(filename,libName);`

This command loads the external library `filename`. Since DLLs can contain malware the user is asked to accept this step. Commands and constants of the DLL may then be used by

`\libName.`*command*`;`

The second argument `libName` has to be a valid class name and must be different from already used class or library identifiers.

Example 8.1 If we want to use a numerical routine `dft` (computing a discrete Fourier transform) that is contained in the library `Numerics.dll`, we write

```
\syst.LoadLibrary(DocDir + "\Numerics.dll", "num");
\num.dft(x);
```

Once a library is imported, it can be used in every document. As soon as the calling document is closed, also the library is removed. Therefore, we remind of the different behavior of the commands `GoToPage`, `ShowDoc`, and `ShowModal`. The first one closes the document while the other two leave the document open.

Functions in DLLs may have arguments or return values of type `Pointer`. Those can be used to execute other functions contained in different libraries. To obtain the correct pointer to a function we use `syst.FunctionPointer`.

Syntax: `\FunctionPointer(libName,functionName)`

This command returns the pointer to the function `functionName` which is contained in a DLL that was previously imported under the identifier `libName`. If no such function exists or the DLL has not been loaded, the return value points to the zero-address.

Example 8.2 Let us have a look at the usage of `FunctionPointer`. The DLL `DLL1.dll` contains the following function

```
double Test(double x) = { return exp(-x) -x; }
```

and provides the function

```
GetTest_VP(int argType[ ], int argSize[ ], void *argPointer[ ])
{
 // check if function was called with the correct arguments
 if ((*argType != DLL_ARG_VOID_POINTER) || (*++argType != 0) ||
       (*++argType != DLL_ARG_NONE))
     return ERROR_ARGUMENT;
 *(void**)(*argPointer) = Test;
 return DLL_RESULT_OK;
}
```

that returns the pointer to the function `Test`. Another DLL `DLL2.dll` provides the function

```
int UseTest_V(int argType[ ], int argSize[ ], void *argPointer[ ])
{
 // check if function was called with the correct arguments
 if ((*argType != DLL_ARG_VOID_POINTER) || (*++argType != 0) ||
       (*++argType != DLL_ARG_NONE))
     return ERROR_ARGUMENT;
 typedef double (*DtoD)(double);
 DtoD f = (DtoD)(*(void**)(*argPointer));
 double x = f(0.0);
 return DLL_RESULT_OK;
}
```

that requires a pointer `argPointer` to a function of special type. Now we write in our SigMath document.

```
\syst.LoadLibrary(DocDir + "\DLL1.dll", "firstDLL");
\syst.LoadLibrary(DocDir + "\DLL2.dll", "secondDLL");

\variable Test = firstDLL.GetTest();
\secondDLL.UseTest(Test);
```

Then the function `Test` will be executed.

8.2 Writing DLLs

In the following we briefly want to describe criteria to write customized DLLs for usage within SigMath. On request the files `ZS_ArgTypesDLL.h`, `ZS_ArgUtDLL.h`, and `ZS_ArgUtComplDLL.h` are available, containing definitions and auxiliary functions.

A function written in C/C++ has to match the following requirements in order to be exported to SigMath.

Syntax:

```
int PrefixDLLIndentifier_ReturnType (int argType[ ],
                      int argSize[ ],void *argPointer[ ])
```

Analogously, constants or properties must have the following form

Syntax:

```
int PrefixDLLPROP_indentifier_ReturnType (int argType[ ],
                           int argSize[ ], void *argPointer[ ])
```

Here `PrefixDLL` is a predefined prefix. By default, the underscore `_` is used. The suffix `ReturnType` represents the data type of the returned value of the function or the data type of the constant in SigMath. All possible suffices are summarized in Table 8.1. Both DLL functions return a value `i` of type `Integer`. If `i=0` (`DLL_RESULT_OK`) no error occurred. If `i` is positive, SigMath calls the function `char *GetText_V(int)` in the DLL (if existing) with argument `i`. If `GetText_V` returns a non-zero pointer, the corresponding text is returned to the user. Otherwise SigMath produces an error message containing the name of the function and DLL and the value of `i`. If `i` is negative, SigMath closes the current document. The function `GetText_V` is not accessible by the user.

Suffix	Return type
_I	Integer
_R	Real
_C	Complex
_B	Bool
_V	Void
_S	String
_VP	Pointer

Tbl. 8.1: Possible Suffices for DLL functions and properties

The arguments of the function have the following meaning: `argType` describes the types of pointers contained in `argPointer`. The argument `argSize` contains sizes of arrays.

As an example, we consider a function which has two arguments. The first argument is a real number:

```
argType[0] = DLL_ARG_DOUBLE
argType[1] = 0   dimension
argSize   no entry
argPointer[0] :  pointer to double
```

The second argument is a real matrix of size 10×20:

```
argType[2] = DLL_ARG_DOUBLE
argType[3] = 2  dimension
argSize[0] = 10
argSize[1] = 20
argPointer[1]:  pointer to double
```

The last entry of `argType` is always `DLL_ARG_NONE`. All pointers in `argPointer` should only be used within this function. On empty arrays, the corresponding entry in `argPointer` is not a valid pointer. For example, if `argPointer[i]` corresponds to an empty array, `argPointer[i]=i+1`. This property is used by SigMath to identify empty arrays. If a DLL function has more than one array as arguments, then the programmer has to make sure, that at most one empty array is possible.

The exported function or property can be used in SigMath as usual. For the above function we write `Identifier(...)`. The prefix `PROP` for properties or constants allows to write `libName.Identifier`, where `libName` is the name of the library used in `LoadLibrary`. SigMath actually calls the function in the DLL that returns the value of the property.

If a function in a DLL changes the size or layout of an array or `String` the DLL must contain a variable of type `RECALL_RESIZE`. The value of the variable can be changed by the function `Message_V` and contains a pointer to the function that implements the method to resize the object. It has the interface

```
int RecallResize (void** dataP, int dim, int sizes[ ])
```

Here, `*dataP` is pointer to the object which should be changed, `dim` its new dimension, and `sizes[ ]` its current layout. The function `RecallResize` can change the value of the variable `*dataP`.

The function `Message_V` must be contained in the DLL. It allows to react on SigMath mes-

sages and is only used internally. A possible implementation is given in the following.

```
__declspec(dllexport)
int Message_V(int type, void *argPointer[ ])
{
 switch(type)
 {
   case DLL_MSG_AFTER_LOAD:
           Resize = (RECALL_RESIZE)argPointer[0];
           further initializations
           break;
   case DLL_MSG_BEFORE_DELETE:
           clean up
           break;
 }
return DLL_RESULT_OK;
}
```

The first argument of the function is the message type, the value of `argPointer` depends on the type. Currently, only two types are implemented.

1. The message `DLL_MSG_AFTER_LOAD` is sent after loading the library;
   `argPointer` contains pointers to the following SigMath functions:

   ```
   argPointer[0]:   Resize
   argPointer[1]:   GetFunction
   argPointer[2]:   FreeFunction
   ```

   Instead of writing `0,1,2` one can also use the following symbols, defined in `ZS_ArgTypesDLL.h` `DLL_INDEX_RECALL_RESIZE`, `DLL_INDEX_GET_FUNCTION`, and `DLL_INDEX_FREE_FUNCTION`.

2. The message `DLL_MSG_BEFORE_DELETE` is sent before removing the library;
   here `argPointer = 0`.

Example 8.3 Let us have a look at an example that changes the layout of an array. The following function writes the content of the file `FileName` to the matrix `D`.

```
__declspec(dllexport)
int LoadArray_V(int argType[ ], int argSize[ ], void *argPointer[ ])
{
 char   *FileName;
 int     Sizes[2];
```

```
 double *D;
 FILE   *fp_in = 0;

 //arguments are the filename and the matrix
 if (Get1CharP1DoubleMatrixP(argType, argPointer, & FileName, & D) != DLL_RESULT_OK)
   return -1;
 fp_in = fopen(FileName, "r");
 if (fp_in == 0)
   return -1;
 // determine the size of the matrix in FileName and write sizes
 // adapt D:
 if (Resize((void**)(&D), 2, Sizes) != DLL_RESULT_OK)
   return -1;
 // fill D
 return DLL_RESULT_OK;
}
```

Calling user-defined functions

If one wants to use a user-defined SigMath function in a DLL we have to generate a pointer to the function. This is done by using the commands `GetFunction` and `FreeFunction`. Currently, only two calls of `GetFunction` are possible without issuing one call of `FreeFunction`. The actual call is performed by `Message_V`. We will use the following definitions contained in `ZS_ArgTypesDLL.h`:

```
typedef int             (*RECALL_FUNCTION)     (int[ ], int[ ], void*[ ]);
typedef RECALL_FUNCTION (*RECALL_GET_FUNCTION) (const char*);
typedef void            (*RECALL_FREE_FUNCTION)(RECALL_FUNCTION);
```

Then, `Message_V` may have the following implementation:

```
RECALL_GET_FUNCTION   GetFunction(0);
RECALL_FREE_FUNCTION  FreeFunction(0);

int Message_V(int type, void *argPointer[ ])
{
 switch(type)
 {
   case DLL_MSG_AFTER_LOAD:
     GetFunction  = (RECALL_GET_FUNCTION) argPointer[DLL_INDEX_GET_FUNCTION];
     FreeFunction = (RECALL_FREE_FUNCTION)argPointer[DLL_INDEX_FREE_FUNCTION];
     break;
   default:
```

```
      break;
  }

  return DLL_RESULT_OK;
 }
```

After setting a pointer `F` by calling `GetFunction` one can use the following function

```
int F (int argType[ ], int argSize[ ], void *argPointer[ ])
```

The arguments have the same meaning as at the beginning of this section. If `argPointer` contains one entry more than the associated user-defined functions, the last entry is used as return value. The following data types are supported:

Arguments	`Real`, `RealArray`, `Integer`, `String`
Return value	`Real`, `RealArray`, `Integer`

Example 8.4 Let us consider the following example. We want to use the function

```
\function f(x) = { return sin(x); }.
```

in our DLL. We write

```
RECALL_FUNCTION F(0);
char *name = "f";
F = GetFunction(name);

double x(1.0), y(0,0);
int ArgType[ ] = { DLL_ARG_DOUBLE, 0, DLL_ARG_DOUBLE, 0, DLL_ARG_NONE};
void *ArgPointer[ ] = { (void*)(&x), (void*)(&y)};

F(ArgType, 0, ArgPointer);

FreeFunction(F);
```

Then `y=sin(1)`.

9 Rendering

Index

(), 85, 94
*, 80, 81
*/, 18
*=, 81, 82
+, 80, 81, 85
++, 81, 95
+=, 81, 82, 85
+INF, 92
-, 80, 81
-=, 81, 82
-INF, 92
/, 80, 81
/*, 18
//, 18
/=, 81, 82
<, 81
<=, 81
=, 81
==, 81, 105
>, 81
>=, 81
%, 18
&, 33
&&, 81, 83
ˆ , 80, 81
`gl.LineWidth`, 129
`gl.PolygonStipple`, 131
`gl.Rotate`, 136
`gl.Scale`, 135
`gl.Translate`, 136
\, 17, 27
\∼, 27
\„ 27
\:, 27
\;, 27
\\, 26, 29, 33
—, 81, 95
\, 77

AngleDegree, 101
AngleRadian, 101
anim, 72
ansi, 24
ANSIToString, 91
arabic, 24
array, 22, 33–35, 44, 84, 113
ArrayBlock, 35, 97
ArrayColSep, 34
ArrayColX, 36
ArrayGetCell, 36, 37
ArrayIndex, 36
ArrayRect, 36
ArrayRowY, 36
Arrays, 75
ArrayStretch, 34
AutoTicks, 60, 64
AxesColor, 60, 64
AxesDepth, 61, 64
AxesRanges, 60
AxisMaxRange, 62
AxisPos, 61
AxisRange, 60, 63

back, 94
Backslash, 85
BaseLineStretch, 26
begin, 105
begin{document}, 16
bf, 25

BgColor, 33
BgColoredRect, 32
BgImage, 32
BINARY, 91
BindPlot, 57
BindPlotItem, 57
Bitmap, 75, 145
bitmap, 84
bmp.LoadPower2, 145
Bool, 75, 78
Brackets, 35
break, 108, 111
BreakPoint, 118
Busywaiting, 73

case, 76, 107, 108
centerline, 22, 26
chinese, 24
clear, 104
code, 17, 20, 77
ColAlign, 35
Color, 32, 33, 124
ColSepDelta, 34
Complex, 75, 78, 82, 90
cont, 84, 113
Container, 75
continue, 111
crArrow, 48
crCross, 48
crDrag, 48
crHandPoint, 48
crHourGlass, 48
crNo, 48
CrossProd, 101
crSizeNESW, 48
crSizeNS, 48
crSizeNWSE, 48
crSizeWE, 48
crUpArrow, 48
CurrentPageLabel, 40
CurrentPageNr, 40
Cursor, 48

DECIMAL, 91
default, 24, 76, 108
Delete, 87
Digits, 42, 61
Dim, 95
discrete textures, 141
DocDir, 18, 19
DocName, 18, 19
document, 22
DoString, 41, 109
DoValueOf, 41

e, 76
edit, 84
else, 76, 106, 108
empty, 95, 104
end, 105
end{document}, 16
endif, 107
enumerate, 22, 29
env, 22, 68
environment, 21
Execute, 20
ExecuteString, 20
EXP10, 90

false, 76
FILL, 30, 55, 57
FILLED_RECT, 55
FillFromString, 95, 96
finite state machine, 121
FIXES_POINT, 90
FloagDigits, 90
FloatDigits, 90
FloatFormat, 90, 91
FlushRight, 25
FocusedInput, 43
fontcharset, 23, 24
fontname, 23
fontpoints, 23, 24
footnotesize, 25
for, 76, 110
FrameNr, 73
FramesPerSec, 73
FreeFunction, 156

front, 94
function, 76, 113

GetAxesRanges, 60
GetBgColor, 33
GetBlock, 96
GetColor, 33
GetFunction, 156
GetGoToColor, 33
GetInputColor, 33
GetInputColorFrame, 33
GetInputColorSelBg, 33
GetInputColorSelText, 33
gl.ALPHA, 131
gl.BACK, 123
gl.BindTexture, 142
gl.CLAMP, 143
gl.COMPILE, 134
gl.COMPILE_AND_EXECUTE, 134
gl.DEPTH_COMPONENT, 131
gl.FILL, 123
gl.FRONT, 123
gl.FRONT_AND_BACK, 123
gl.Frustum, 140
gl.GenList, 134
gl.GenTextures, 142
gl.Get, 130
gl.GL_Q, 148
gl.GL_R, 148
gl.GL_S, 148
gl.GL_T, 148
gl.IsList, 135
gl.LINE, 123
gl.LINE_LOOP, 125
gl.LINE_STRIP, 125
gl.LINEAR, 143
gl.LINEAR_MIPMAP_LINEAR, 143
gl.LINEAR_MIPMAP_NEAREST, 143
gl.LINES, 124
gl.LineStipple, 130
gl.LUMINANCE, 131
gl.MODELVIEW, 136
gl.NEAREST, 143
gl.NEAREST_MIPMAP_LINEAR, 143
gl.NEAREST_MIPMAP_NEAREST, 143
gl.NewList, 134
gl.POINT, 123
gl.POINTS, 124
gl.POLYGON, 129
gl.PopMatrix, 138
gl.PROJECTION, 136
gl.PushMatrix, 138
gl.QUAD_STRIP, 128
gl.QUADS, 128
gl.REPEAT, 143
gl.RGB, 145
gl.TexGen, 149
gl.TexImage2D, 142
gl.TexParameter, 143
gl.TEXTURE, 136
gl.TEXTURE_2D, 143
gl.TEXTURE_MAG_FILTER, 143
gl.TEXTURE_MIN_FILTER, 143
gl.TEXTURE_WRAP_S, 143
gl.TEXTURE_WRAP_T, 143
gl.TRIANGLE_FAN, 127
gl.TRIANGLE_STRIP, 127
gl.TRIANGLES, 126
gl2ps.FontName, 67
GL_TEXTURE_GEN_MODE, 148
GLPoint, 75, 99
glpoint, 84, 113
GLPointString, 91
glu, 121
glu.Build2DMipMaps, 142
glu.Cylinder, 132
glu.Disk, 132
glu.FILL, 133
glu.LINE, 133
glu.LookAt, 139
glu.OUTSIDE, 134
glu.PartialDisk, 133
glu.Perspective, 140
glu.POINT, 133
glu.QuadricDrawStyle, 133
glu.QuadricTexture, 148

glu.SILHOUETTE, 133
glu.Sphere, 132
GoToColor, 33, 40
GoToPage, 39, 151
greek, 24

Height, 72
HelpDir, 18, 19
HEXADECIMAL, 91
hline, 33
HSpace, 27
Huge, 25
huge, 25

I, 76, 78
if, 76, 78, 106
IgnoreIf, 107
Image, 30–32, 57
ImageStretch, 30, 32
Include, 18–21, 65
IncludeString, 19, 20
INF, 43, 76
Input, 42, 43, 100
InputArrayBlock, 35, 43, 44, 97
InputColor, 33, 41
InputColorFrame, 33
InputColorSelBg, 33
InputColorSelText, 33
InputDelta, 42
inputlabel, 22, 40
InputMax, 42
InputMin, 42
Insert, 87
insert, 98, 105
Integer, 75, 78, 81–83, 87
Interactive, 62, 64
IntFormat, 90
IsAlNum, 86
IsAlpha, 86
IsDigit, 86
IsKeyDown, 46
IsLeftButtonDown, 48
IsRightButtonDown, 48
IString, 90

it, 25
item, 29, 30

label, 39, 65
LARGE, 25
Large, 25
large, 25
LeftMargin, 23
Length, 101
LINE_STRIP, 55
linebreak, 26
List, 103
Load, 88, 97
LoadFromClipboard, 89
LoadFromKey, 89
LoadTexture, 142

M_AFTER_CHANGE, 44
M_BEFORE_CHANGE, 44
M_BEFORE_DRAW, 44
M_IS_INPUT_ENTRY, 44
M_MOUSE_DOWN, 44
M_MOUSE_UP, 44
Map, 103
Mean, 99
Message_V, 154
MIXED, 90
monospace, 24, 67
MouseDeltaX, 47
MouseDeltaY, 47
MouseX, 47
MouseY, 47
MultiMap, 103
MultiSet, 103
Multiset, 103

n, 86
NAN, 76, 92
newline, 26
newpage, 39, 65
newpage;, 65
nextlevel, 30
NONE, 57
normalsize, 25

OCTAL, 91
OnDocClose, 45
OnPageChange, 45
OnVarChange, 45
OnWinMessage, 45
OpenGL, 121
overline, 25

PageLabel, 41
ParentDir, 18, 19
Pi, 76
Plot, 51, 55, 56
plot2d, 22, 49, 51, 52, 57
plot3d, 22, 51
PlotAxes, 60, 64
PlotBgColor, 56, 57, 64
PlotBgStyle, 56, 64
PlotColor, 55, 64
PlotDepth, 56, 64
PlotHeight, 52, 64
PlotIndex, 36, 57
PlotLineStipple, 55, 64
PlotLineStipple(), 56
PlotLineWidth, 55, 64
PlotMargins, 57, 64
PlotNumPointsX2D, 51, 64
PlotNumPointsX3D, 64
PlotNumPointsY3D, 64
PlotPointSize, 56, 64
PlotPos, 52
PlotReDraw, 57
PlotStyle, 64
PlotTick, 60
PlotWidth, 52, 64
PlusString, 89, 91
POINT, 55
Pointer, 75
Pointer(), 79
PolygonMode, 123
pop_back, 99, 105
pop_front, 105
Position, 86
PositionCI, 86
postext, 22, 28, 63
PrefixDLL, 153
prevlevel, 30
primitives, 121
procedural textures, 141
ProgDir, 18, 19
PROP, 30
PROP_MAX_2, 30
push_back, 99, 105
push_front, 105

qquad, 27
quad, 27, 35
Quote, 85

Ratio, 72
Real, 75, 78, 81, 82
RECT, 55, 57
Remove0s, 90
rendering, 122
Replace, 87
Resize, 97
return, 113
reverse, 98, 99
rotation, 136
russian, 24

Save, 88, 97
SaveToClipBoard, 89
SaveToKey, 88
Scaling, 135
scenery, 138
ScreenHeight, 72
ScreenWidth, 72
scriptsize, 25
SelectedDo, 41
selectfont, 23, 25
SelObj, 48, 49
SelObjInd, 49, 50
SelObjMessage, 49
Set, 102, 103
SetBlock, 96
SetMouseXY, 48
SetVersion, 95

Size, 85
size, 95, 104
small, 25
SquaredLength, 101
String, 75, 79, 84, 85, 89
string, 84, 85, 89, 113
SubString, 86
SubstringBefore, 86
Sum, 99
swap, 105
switch, 76, 107
symbol, 24
syst.FunctionPointer, 151
syst.LoadLibrary, 151

t, 86
TexImage2D, 144
TextDepth, 29
TextEditor, 75
TextPos, 28, 63
TextPosPixel, 28
TextPosPlotRect, 63
texture mapping, 141
TextureFromString, 146
TextureLoad, 142
textures, 141
TextWidth, 23
thread.BeginThread, 114
thread.EnterCriticalSection, 115
thread.InitializeCriticalSection, 115
thread.LeaveCriticalSection, 115
TickLengthFactor, 61, 64
Time, 73
tiny, 25
ToInt, 91, 92
ToLower, 87
TopMargin, 23
ToReal, 91, 92
ToString, 89, 97
ToUpper, 87
transformation, 135
translations, 136
Transpose, 98
Trim, 88
true, 76, 78
TrueString, 91
TypeOf, 76

underline, 25
use_class, 76
use_class className, 71

ValueOf, 77, 94, 100
variable, 75, 76
Version, 95
vertex, 122
Vertex(), 122
VK_CONTROL, 46
VK_DOWN, 46
VK_ESCAPE, 46
VK_F1, 46
VK_LEFT, 46
VK_NEXT, 46
VK_PRIOR, 46
VK_RIGHT, 46
VK_SHIFT, 46
VK_SPACE, 46
VK_UP, 46
Void, 75
VSpace, 27

w, 100
while, 76, 111
Width, 72
WM_KEYDOWN, 46
WM_KEYUP, 46
WM_LBUTTONDOWN, 46, 49
WM_LBUTTONUP, 46, 49
WM_MOUSEMOVE, 46, 49
WM_RBUTTONDOWN, 46
WM_RBUTTONUP, 46
WM_SYSKEYDOWN, 46
WM_SYSKEYUP, 46

x, 100

y, 100

z, 100